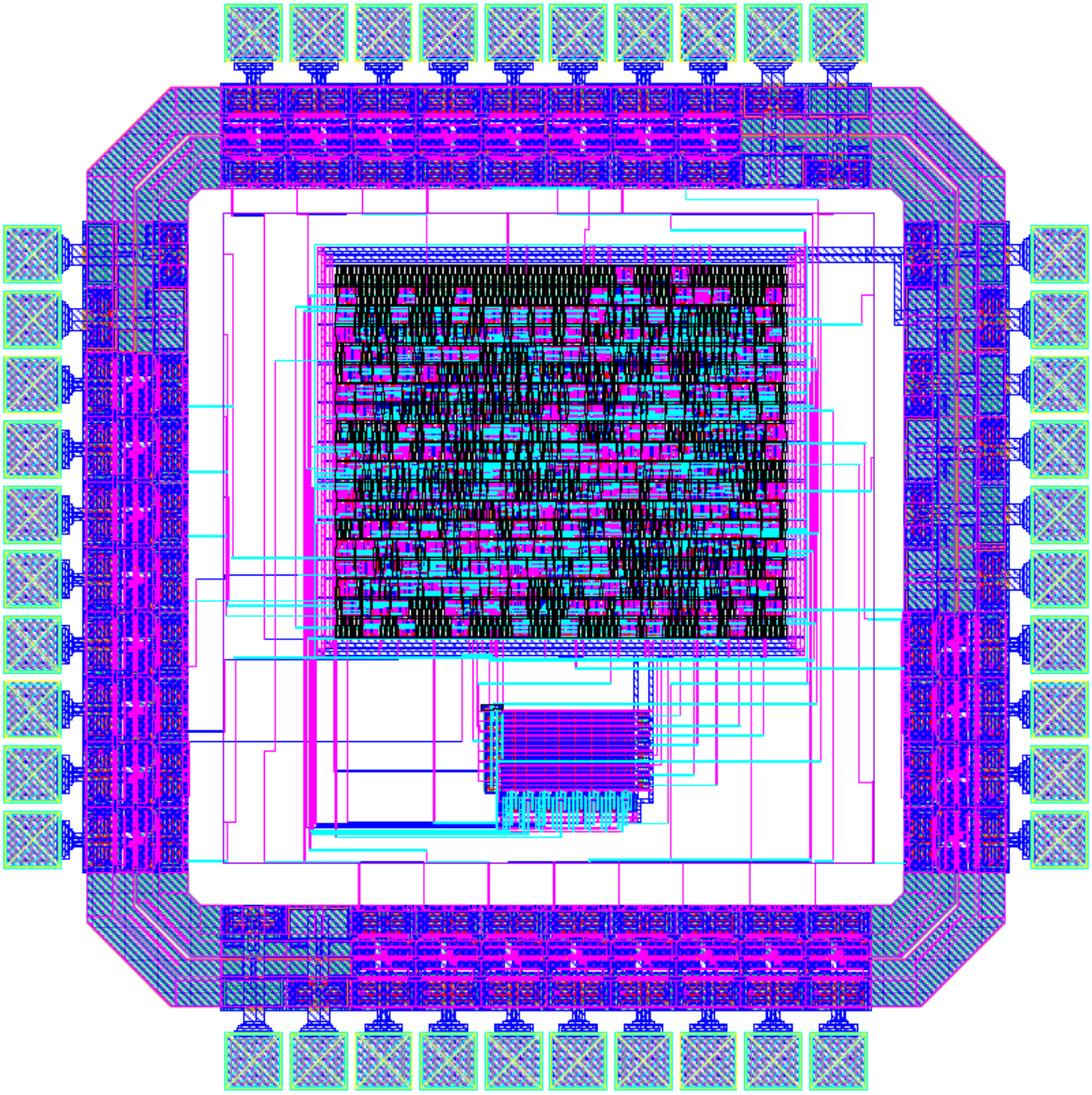


Conway's Game of Life

Narayan Propato
Nicholas Card

Contents

Appendix

Introduction

The goal of this project is to implement Conway's Game of Life on a 1.5 x 1.5 mm 40-pin MOSIS "TinyChip" fabricated in a 0.6 μm process. Conway's Game of Life consists of a matrix (8 by 8 in this project) consisting of dead and live cells. The user is able to provide a starting matrix, and then the chip will calculate subsequent iterations of the matrix according to a set of rules and output this data.

Specifications

The user is able to input rows of data making up the initial state of the 8 by 8 board. The user interface comprises a reset button, an enter button, and eight switches. Each switch corresponds to a cell in a row (0 for dead and 1 for alive), while the enter button is used to input the set of values on the switches into the currently selected row. When all row data has been entered, each press of the button causes the next iteration of the board to be calculated according to the following rules:

1. A live cell with fewer than 2 adjacent live cells dies.
2. A live cell with more than 3 adjacent live cells dies.
3. A live cell with 2 or 3 adjacent live cells continues to live.
4. A dead cell with exactly 3 adjacent live cells becomes a live cell ("Conway").

The reset button is used to start a new game. The board information is stored in an SRAM memory block and outputted via the ledpower[7:0] pins and ledcolumn[7:0] pins. The chip uses time multiplexing to quickly cycle power between the 8 ledpower signals, each of which will be connected to a row of an 8 by 8 dot-matrix LED display. The 8 ledcolumn signals will be connected to the columns of the display representing the values in the row that is

currently being powered. See Table 1 for a table of all the pins on the chip. The project uses 34 out of the 40 pins: 12 input pins, 16 output pins, and 6 bi-directional pins.

Input Pins	Output Pins
reset	power [7:0]
enter	column [7:0]
switch [7:0]	
ph1, ph2	
VDD/GND [5:0]	

Table 1. Table lists all 40 pins used in the design. Note that only one pair of VDD and GND pins are actually connected to the chip's core.

Floorplan

The proposed floorplan covered six modules: the *controller*, the *register*, the *bit selector*, the *hold mem*, the *output logic*, and the *temp mem*, as seen in Fig. A1.1(a). Other than the register, all modules would be automatically synthesized from Verilog modules. The register would be custom designed, including the *wordline* conditioning logic and the *bitline* conditioning logic (see the slice plan in Fig. A1.2). During further design iterations, all synthesized modules were instanced within a single top-level module, *gameoflife*, to optimize synthesis (see Fig. A1.1(b) for final floorplan and Fig. A1.3 for pinout diagram).

Originally, 4.56 $k\lambda^2$ was budgeted for all synthesized blocks combined, and the final gameoflife block uses 4.34 $k\lambda^2$, which is 4.9% smaller than expected. This difference is primarily due to the uncertainty in estimating the original controller size.

The custom block takes up 0.47 $k\lambda^2$, which is 2.15 times the allotted area of 0.22 $k\lambda^2$. Originally, the register used 6-transistor SRAM cells (Harris 504), but the extra design time required to properly adjust the width-ratio between the bitline conditioning logic and the memory cell proved prohibitive. The cell was thus changed to a 12-transistor version (Harris 506) during subsequent design iterations in order to reduce the design time burden. The area difference

between the proposed and final registers is due to the 12-transistor cell being approximately twice the size of the 6 transistor cell, as well as the extra wiring tracks needed for the extra signals in the 12-transistor cell.

Verification

The Verilog code (see A3.1) passes the testbench on the chosen test vectors (see A4 for testing materials). The schematic (including the register and synthesized logic) also passes the testbench. The layout passes DRC and LVS, confirming that it matches the schematic and therefore passes the testbench.

There is one caveat: due to an undetermined error in the synthesis process, an enable/reset flip-flop synthesizes in part to a mux that loops its output to its select signal. Since the output starts as a conflict `x`, the mux never resolves the output. This means that any module with an enable/reset flip-flop will not simulate correctly. To fix this, the output of the latch in the flip-flop must be set to `1` on start. The netlist used to do this is in A3.2.

Postfabrication Test Plan

After fabrication, the chip's VDD and GND pins (pins 5 and 4, respectively) will be connected to a 5 V power supply. The ph1 and ph2 pins will be connected to the appropriate clock sources. The “enter” and “reset” pins will be connected to 1-bit buttons that pass `1` on press. The “switch” pins will be connected to an 8-position DIP switch such as Tyco Electronics’ ADE0804 switch. Finally, the output pins will be connected to a dot-matrix LED display such as Avago Technologies' HDSP -S80E/S80G. The “ledpower” pins will be connected to the rows of the display, and the “ledcolumn” pins will be connected to the columns, such that ledpower[7] is connected to the top row and ledcolumn[7] is connected to the left-most

column. Once the circuit is assembled, testing will consist of inputting a test matrix, pressing the enter button to cause the chip to calculate new iterations, and then comparing the resulting matrix with the expected one. If the two match, the chip is fully functional.

Design Time

A breakdown of design time by category is shown in Fig. 1. Coding the Verilog and debugging it for synthesis made up 72% of the man-hours spent on the project.

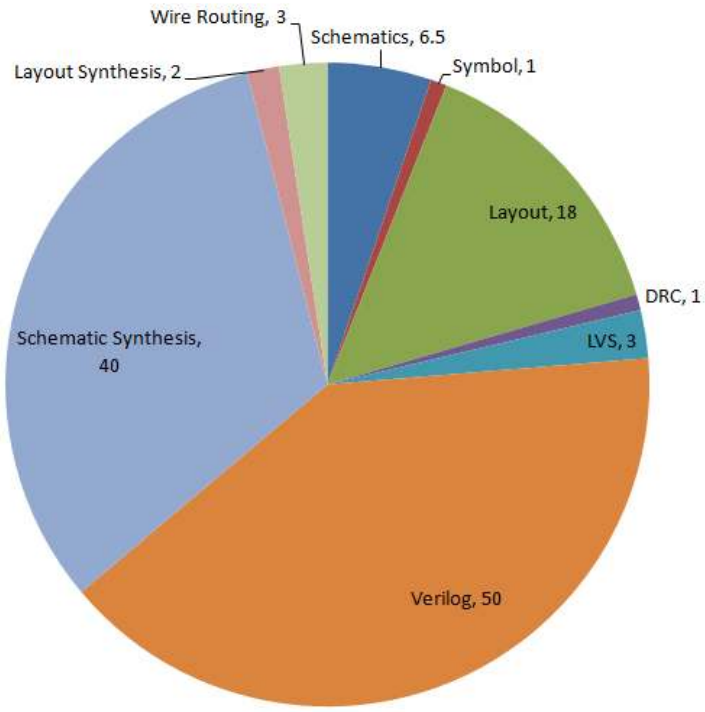


Fig. 1. A breakdown of design time for different design stages in man-hours.

References

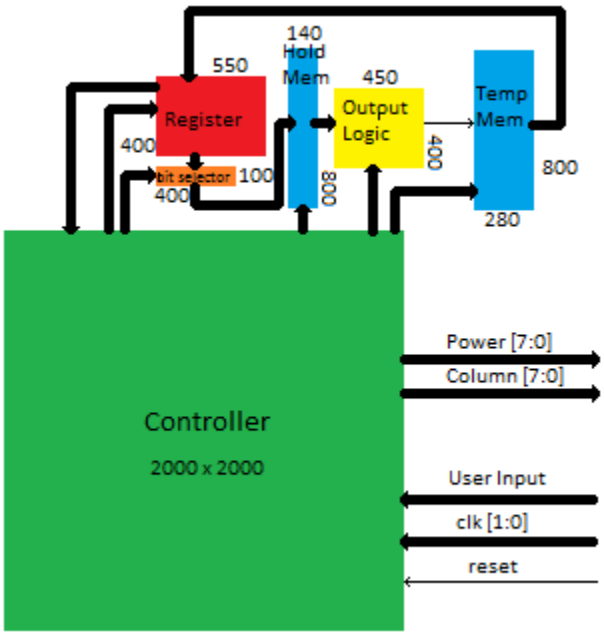
"Conway's Game of Life." *LifeWiki*. Retrieved from http://www.conwaylife.com/wiki/index.php?title=Conway%27s_Game_of_Life.

Harris, David Money and Weste, Neil H. E. *CMOS VSLI Design*. Boston: Addison-Wesley, 2011.

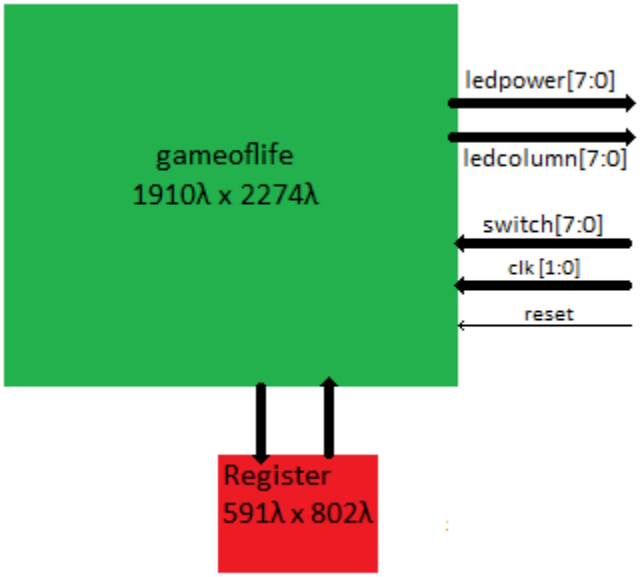
Appendix 1:

Floorplan

A1.1: Floorplan



(a)



(b)

Figure A1.1. The floorplans from the proposal (a) and from the actual chip (b). The *gameoflife* module contains the *controller*, the *bit selector*, the *output logic*, the *temp mem*, and the *hold mem*.

A1.2: Array Plan

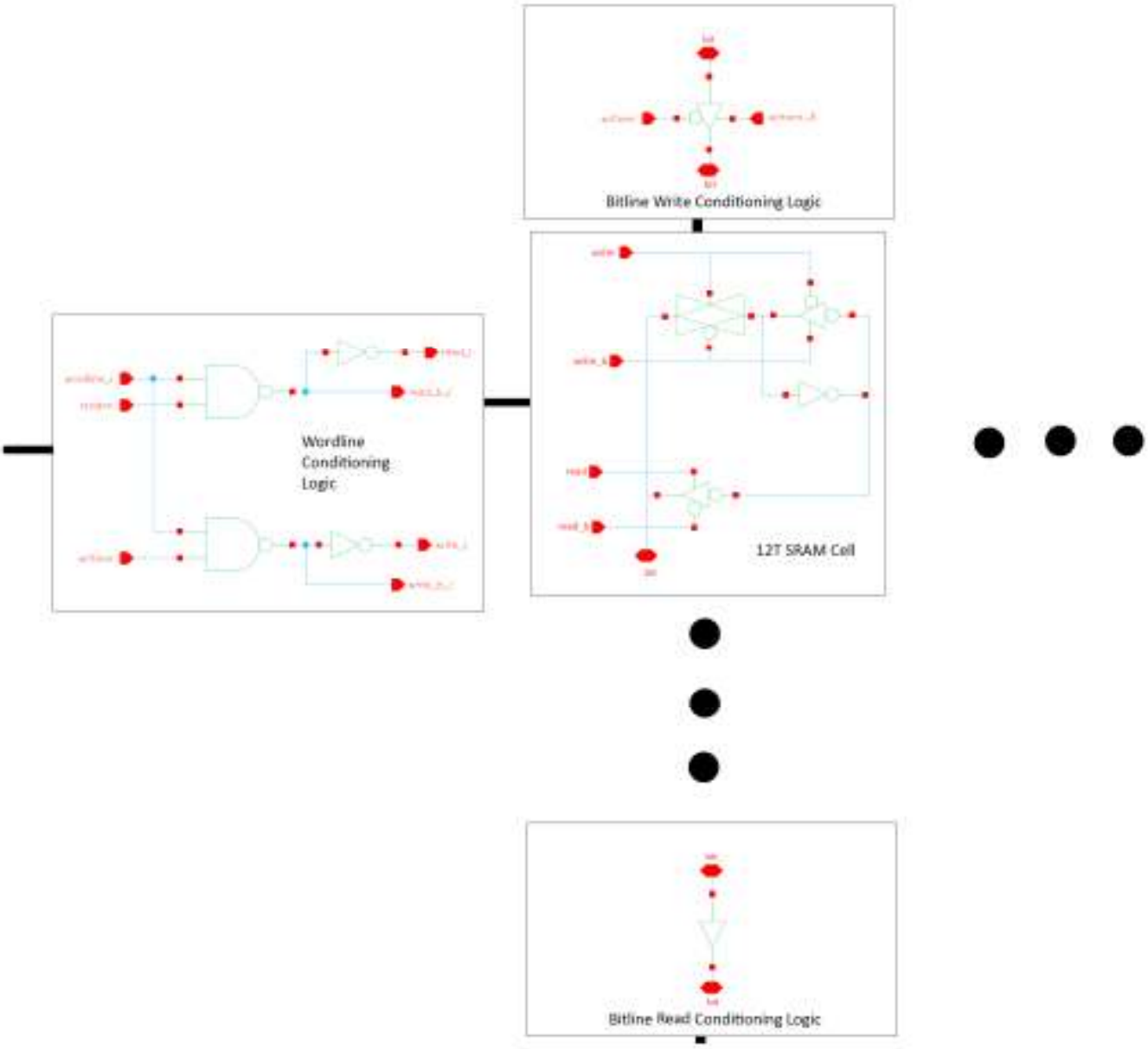


Fig A1.2. An Array Slice from the custom logic. The complete array is 8 x 8.

A1.3: Pinout

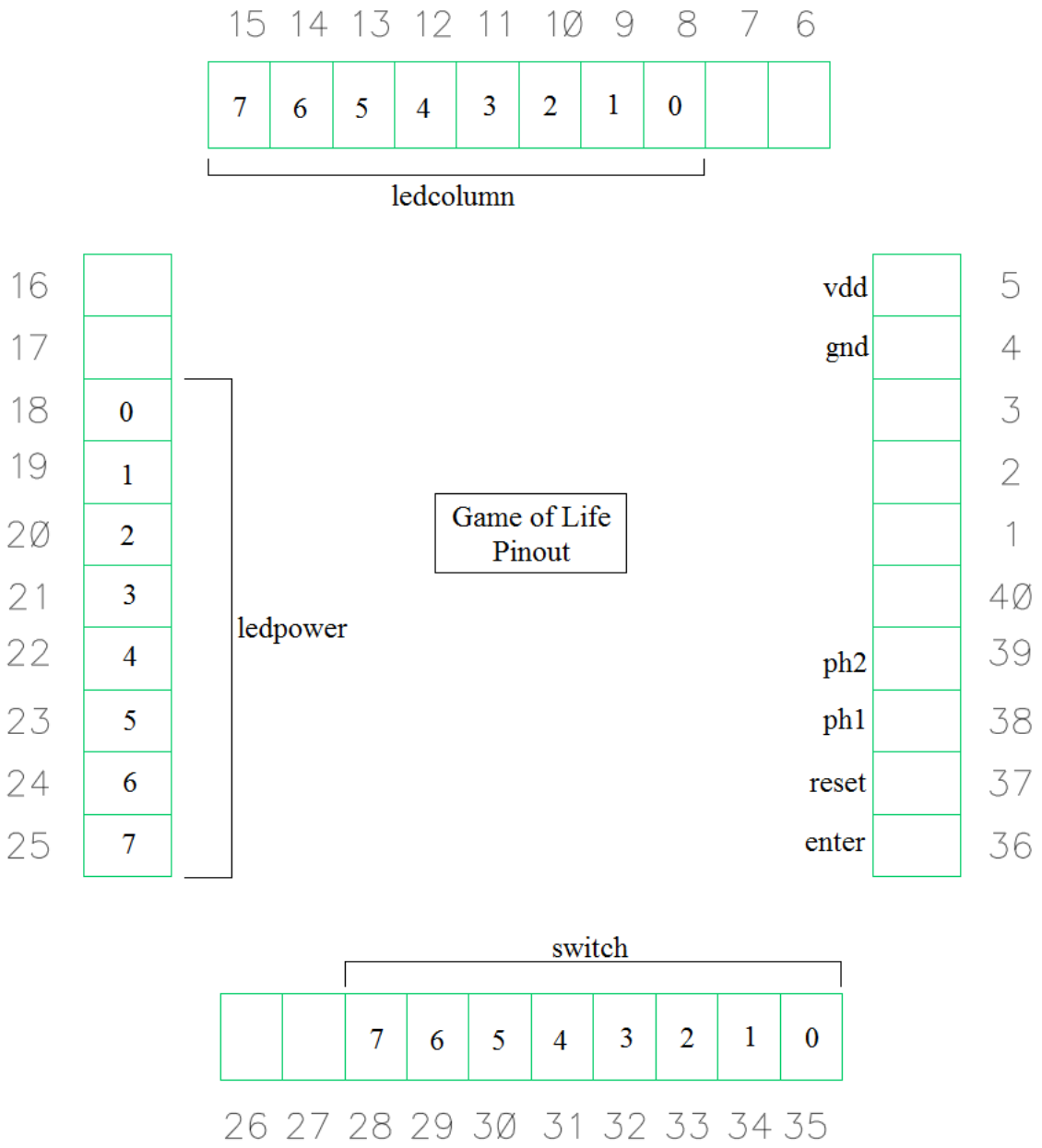


Figure A1.3. The pinout for the Game of Life cell. The number in the pins represents the bit of the signal.

Appendix 2:

File Locations

A2: File Locations

File	Location on Chips (chips.eng.hmc.edu)
Verilog Code	/home/ncard/gameoflife/gameoflife.v
Test Fixture	/home/ncard/gameoflife/verilogtestfixture.v
Test Vectors	/home/ncard/gameoflife/Testmatrixcompilation.txt
Synthesis Results	/home/ncard/gameoflife/gameoflife_syn.v
Cadence Libraries:	/home/ncard/IC_CAD/cadence/
project2_nc (contains custom blocks)	/home/ncard/IC_CAD/cadence/project2_nc/
gameoflife (contains chip and synthesized blocks)	/home/ncard/IC_CAD/cadence/gameoflife/
CIF	/home/ncard/gameoflife/chip.cif
PDF Chip Plot	/home/ncard/gameoflife/chip.pdf
PDF Copy of This Report	/home/ncard/gameoflife/report.pdf

Appendix 3:

Verilog Code

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Company: Harvey Mudd College
4   // Engineer: Narayan Propato
5   // Create Date:    16:47:42 03/21/2010
6   // Design Name: Conway's Game of Life
7   // Module Name:    gameoflife
8   // Project Name: Conway's Game of Life
9   // Description: This is the Verilog module that will interface with the 64-bit
10  //  memory module to make up the core of the chip. The module takes in the user
11  //  inputs reset, enter, and switch and outputs ledpower and ledcolumn off the
12  //  chip. It interfaces with the memory via bitlineread, bitlinewrite, wordline,
13  //  writeen and readen. Note that core and the memory instance within it are only
14  //  used for behavioral simulation. The actual chip only uses the gameoflife
15  //  module and a custom-made memory. Please note that the modules flop, flopen,
16  //  flopenr, latch, mux2, and mux3 were written by Professor Harris.
17  //////////////////////////////////////////////////////////////////////////////////
18
19  // The core module is made up of the synthesizable module gameoflife and the
20  // non-synthesizable module memory. The latter is only used for running
21  // behavior simulations on the Verilog.
22  module core(input ph1,ph2,
23              input reset,
24              input enter,
25              input [7:0] switch,
26              output [7:0] ledpower,
27              output [7:0] ledcolumn);
28
29     wire [7:0] bitlineread,bitlinewrite;
30     wire readen,writeen;
31     wire [7:0] wordline;
32
33  gameoflife gameoflife1(ph1,ph2,reset,enter,switch,bitlineread,ledpower,
34                         ledcolumn,writeen,readen,wordline,bitlinewrite);
35  memory memory1(ph1, ph2, reset, writeen,readen, wordline, bitlinewrite,
36                 bitlineread);
37  endmodule
38
39  // The gameoflife module is the synthesizable module that will be used in the
40  // final design.
41  module gameoflife(input ph1,ph2,
42                    input reset,
43                    input enter,
44                    input [7:0] switch,
45                    input [7:0] bitlineread,
46                    output [7:0] ledpower,
47                    output [7:0] ledcolumn,
48                    output writeen,
49                    output readen,
50                    output [7:0] wordline,
51                    output [7:0] bitlinewrite);
52
53     wire [7:0] row2;
54     wire finalrowpipe;
55     wire leden;
56     wire [7:0] column;
57     wire zerorow;
58     wire threebitshift;
59     wire newbiten;
60     wire [2:0] threebits;
61     wire cellbit;
```

```
62          wire [7:0] eightbits;
63          wire newcellvalue;
64          wire [7:0] nledpower;
65          wire [7:0] nledcolumn;
66
67          flopenr #8 ledpowerreg(ph1,ph2,reset,leden,nledpower,ledpower);
68          flopenr #8 ledcolumnreg(ph1,ph2,reset,leden,nledcolumn,ledcolumn);
69
70          controller controller1(ph1,ph2,reset,enter,switch,bitlineread,row2,
71                                 finalrowpipe,leden,nledpower,nledcolumn,
72                                 writeen,readen,wordline,bitlinewrite,
73                                 column, zerorow, threebitshift,newbiten);
74          bitselector bitselector1(column, zerorow, bitlineread, threebits);
75          holdmem holdmem1(ph1,ph2,reset,threebitshift,threebits,cellbit,eightbits);
76          outputlogic outputlogic1(cellbit, eightbits,newcellvalue);
77          tempmem tempmem1(ph1,ph2,reset,newbiten,finalrowpipe,newcellvalue,row2);
78      endmodule
79
80      // The controller is comprised of a finite state machine that handles user
81      // input, writes to and reads from memory, matrix iteration calculation, and
82      // matrix output (to an LED display). The controller first writes 8'b00000000
83      // to each address in the memory, resetting it. It then writes the values in
84      // switch to a row in memory each time the user presses enter. The controller
85      // outputs the matrix at all times, so the user is able to see the rows he or
86      // she has inputted throughout the input process. After the user presses the
87      // enter button for the ninth time, the controller will calculate the new matrix
88      // iteration and display it. It will continue doing this until the reset button
89      // is pressed, at which point the memory will be reset and user input will be
90      // accepted once again.
91      module controller(input ph1,ph2,
92                        input reset,
93                        input enter,
94                        input [7:0] switch,
95                        input [7:0] bitlineread,
96                        input [7:0] row2,
97                        output finalrowpipe,
98                        output reg leden,
99                        output reg [7:0] nledpower,
100                       output reg [7:0] nledcolumn,
101                       output writeenph2,
102                       output reg readen,
103                       output reg [7:0] wordline,
104                       output [7:0] bitlinewrite,
105                       output reg [7:0] column,
106                       output reg zerorow,
107                       output reg threebitshift,
108                       output newbiten);
109
110         wire en;
111         assign en = 1'b1;
112
113         //enter logic
114         // When the user presses enter, the value is held in a register
115         // until the input is processed.
116         wire enteren;
117         wire enterpipe;
118         reg checkedinput;
119
120         flopenr #1 enterreg(ph1,ph2,reset,enteren,enter,enterpipe);
121         assign enteren = ~enterpipe|checkedinput;
122
```

```
123         // userinputover logic
124         // Once the user sets the initial conditions, each subsequent press
125         // of the enter button will cause the matrix to be updated with a new
126         // iteration.
127         wire uioen;
128         reg userinputover;
129         wire uiopipe;
130
131         flopenr #1 userinputoverreg(ph1,ph2,reset,uioen,userinputover,uiopipe);
132         assign uioen = ~uiopipe;
133
134         // writeen logic
135         // To prevent glitches in the write signal which would corrupt the memory
136         // data, writes are only performed while ph2 is high and all signals are
137         // stable.
138         reg writeen;
139         assign writeenph2 = writeen & ph2;
140
141         // State Register
142         wire [4:0] state;
143         reg [4:0] nextstate;
144
145         flopenr #5 statereg(ph1,ph2,reset,en,nextstate,state);
146
147         wire iden;
148         reg iterationdone;
149         wire idpipe;
150         reg displayloop;
151
152         flopenr #1 iterationdonereg(ph1,ph2,reset,iden,iterationdone,idpipe);
153         assign iden = ~idpipe|displayloop;
154
155         reg nnewbiten;
156
157         flopenr #1 newbitenreg(ph1,ph2,reset,en,nnewbiten,newbiten);
158
159         reg nstorerow;
160         flopenrval #1 storerowreg(ph1,ph2,reset,en,1'b1,nstorerow,storerow);
161
162         reg finalrow;
163         flopenr #1 finalrowreg(ph1,ph2,reset,en,finalrow,finalrowpipe);
164
165         // FSM Next State Logic
166         always@(*)
167            casez (state)
168               5'b00000: nextstate = 5'b00001;
169               5'b????1: if (wordline[0])
170                           nextstate = 5'b00010;
171                         else
172                           nextstate = 5'b00001;
173               5'b???1?: if (displayloop&enterpipe)
174                           if (uiopipe)
175                              nextstate = 5'b01000;
176                           else
177                              nextstate = 5'b00100;
178                         else
179                           nextstate = 5'b00010;
180               5'b??1??: nextstate = 5'b00010;
181               5'b?1???: if (storerow) nextstate = 5'b10000;
182                         else nextstate = 5'b01000;
183               5'b1????: if (finalrow) nextstate = 5'b10000;
```

```
184                         else if (idpipe) nextstate = 5'b00010;
185                         else nextstate = 5'b01000;
186                default: nextstate = 5'bxxxxx;
187             endcase
188
189          // FSM Output Logic
190          wire [7:0] wordlinecounter;
191          reg [7:0] nwordlinecounter;
192          wire [7:0] columncounter;
193          reg [7:0] ncolumncounter;
194          wire [9:0] rowcounter;
195          reg [9:0] nrowcounter;
196          wire [7:0] rowinput;
197          reg [7:0] nrowinput;
198          reg colcounteren;
199          reg rowcounteren;
200          reg rowinputreset;
201
202          flopenrval #8 wordlinecounterreg(ph1,ph2,reset,en,8'b10000000,
203                                           nwordlinecounter, wordlinecounter);
204          flopenrval #8 columncounterreg(ph1,ph2,reset,colcounteren,8'b10000000,
205                                         ncolumncounter, columncounter);
206          flopenrval #10 rowcounterreg(ph1,ph2,reset,rowcounteren,10'b1000000000,
207                                       nrowcounter, rowcounter);
208
209          reg rowinputen;
210          reg [7:0] ncolumn;
211
212          flopenrval #8 rowinputreg(ph1,ph2,rowinputreset,rowinputen,8'b10000000,
213                                    nrowinput, rowinput);
214
215          reg [3:0] ncounter1;
216          wire [3:0] counter1;
217
218          flopenr #4 counter1reg(ph1,ph2,reset,en,ncounter1,counter1);
219
220          reg threerowsreset;
221          reg threerowsen;
222          wire [2:0] threerows;
223          reg [2:0] nthreerows;
224
225          flopenrval #3 threerowsreg(ph1,ph2,threerowsreset,threerowsen,3'b100,
226                                     nthreerows, threerows);
227
228          reg [7:0] bitlinez;
229          reg writenewrow;
230
231          always@(*)
232             begin
233             writeen = 0;
234             readen = 0;
235             leden = 0;
236             threebitshift = 0;
237             nnewbiten = 0;
238             bitlinez = 0;
239             wordline = 0;
240             nledpower = 0;
241             nledcolumn = 0;
242             displayloop = 0;
243             checkedinput = 0;
244             nwordlinecounter = wordlinecounter;
```

```
245             ncolumncounter = columncounter;
246             nrowcounter = rowcounter;
247             nrowinput = rowinput;
248             nthreerows = threerows;
249             ncounter1 = counter1;
250             userinputover = 0;
251             rowinputen = 0;
252             colcounteren = 0;
253             rowcounteren = 0;
254             zerorow = 0;
255             threerowsen = 0;
256             iterationdone = 0;
257             writenewrow = 0;
258             nstorerow = 0;
259             rowinputreset = 0;
260             finalrow = 0;
261             threerowsreset = 0;
262
263                   case (state)
264                   // Reset state
265                   5'b00000: begin rowinputreset = 1;
266                             threerowsreset = 1;
267                             nwordlinecounter = wordlinecounter; end
268                   // Memory reset state
269                   5'b00001: begin nwordlinecounter = {wordlinecounter[0],
270                                                       wordlinecounter[7:1]};
271                            wordline = wordlinecounter;
272                            writeen = 1'b1;
273                            bitlinez = 8'b00000000; end
274
275                   // Display state
276                   5'b00010: begin
277                               ncounter1 = counter1 + 1;
278                               if (counter1 == 4'b1111)
279                                  begin
280                                     nwordlinecounter = {wordlinecounter[0],
281                                                         wordlinecounter[7:1]};
282                                     if (wordlinecounter[0])
283                                        begin
284                                        displayloop = 1;
285                                        end
286                                  end
287                               else
288                                  begin
289                                     wordline = wordlinecounter;
290                                     readen = 1'b1;
291                                     leden = 1'b1;
292                                     nledcolumn = bitlineread;
293                                     nledpower = wordlinecounter;
294                                  end
295                             end
296
297                   // Write user input to memory state
298                   5'b00100: begin
299                                if (rowinput[0]) userinputover = 1;
300                                checkedinput = 1;
301                                nrowinput = {rowinput[0],rowinput[7:1]};
302                                wordline = rowinput;
303                                writeen = 1'b1;
304                                rowinputen = 1'b1;
305                                bitlinez = switch;
```

```
306                             end
307
308                     // Calculate new matrix state
309                     5'b01000: begin
310                                 if (rowinput[7]) checkedinput = 1;
311                                 threerowsen = 1;
312                                 nthreerows = {threerows[0],threerows[2:1]};
313                                 column = columncounter;
314                                 threebitshift = 1;
315                                 readen = 1;
316                                 if ((rowinput[7]&threerows[2])|(rowinput[0]&threerows[0]))
317                                    zerorow = 1;
318                                 if (threerows[0])
319                                    begin
320                                       nnewbiten = 1;
321                                       colcounteren = 1;
322                                       ncolumncounter = {columncounter[0],
323                                                         columncounter[7:1]};
324                                       if (columncounter[0])
325                                          begin
326                                             if (rowinput[0]) iterationdone = 1;
327                                             if (~rowinput[7]) nstorerow = 1;
328                                             else
329                                                begin
330                                                   rowinputen = 1;
331                                                   nrowinput = {rowinput[0],rowinput[7:1]};
332                                                end
333                                          end
334                                    end
335                                 casez (threerows)
336                                    3'b1??: wordline = {rowinput[6:0],rowinput[7]};
337                                    3'b?1?: wordline = rowinput;
338                                    3'b??1: wordline = {rowinput[0],rowinput[7:1]};
339                                    default: wordline = 3'bxxx;
340                                 endcase
341                               end
342                     5'b10000: begin
343                                 threerowsreset = 1;
344                                 writeen = 1;
345                                 wordline = {rowinput[6:0],rowinput[7]};
346                                 writenewrow = 1;
347                                 if (rowinput[0]) finalrow = 1;
348                                 if (~rowinput[7])
349                                    begin
350                                       rowinputen = 1;
351                                       nrowinput = {rowinput[0],rowinput[7:1]};
352                                    end
353                                 else if (rowinput[6]&idpipe) rowinputreset = 1;
354                               end
355                endcase
356             end
357
358          assign bitlinewrite = writenewrow ? row2 : bitlinez;
359       endmodule
360
361       // Temporary module used for behavioral simulation purposes. Since the memory
362       // will ultimately be custom-made and not synthesized, this module can be ignored
363       // in the final design.
364       module memory(input ph1, ph2,
365                     input reset,
366                     input writeen,readen,
```

```
367                     input [7:0] wordline,
368                     input [7:0] bitlinewritez,
369                     output [7:0] bitlineread);
370
371           wire [7:0] row1,row2,row3,row4,row5,row6,row7,row8;
372           reg [7:0] bitlinereadz;
373           wire [7:0] bitlinewrite;
374
375           assign bitlinewrite = writeen ? bitlinewritez : 8'bzzzzzzzz;
376
377           flopenrval #8 row1reg1(ph1,ph2,reset,wordline[7]&writeen,
378                                  8'b10000000,bitlinewrite,row1);
379           flopenrval #8 row2reg1(ph1,ph2,reset,wordline[6]&writeen,
380                                  8'b01000000,bitlinewrite,row2);
381           flopenrval #8 row3reg1(ph1,ph2,reset,wordline[5]&writeen,
382                                  8'b00100000,bitlinewrite,row3);
383           flopenrval #8 row4reg1(ph1,ph2,reset,wordline[4]&writeen,
384                                  8'b00010000,bitlinewrite,row4);
385           flopenrval #8 row5reg1(ph1,ph2,reset,wordline[3]&writeen,
386                                  8'b00001000,bitlinewrite,row5);
387           flopenrval #8 row6reg1(ph1,ph2,reset,wordline[2]&writeen,
388                                  8'b00000100,bitlinewrite,row6);
389           flopenrval #8 row7reg1(ph1,ph2,reset,wordline[1]&writeen,
390                                  8'b00000010,bitlinewrite,row7);
391           flopenrval #8 row8reg1(ph1,ph2,reset,wordline[0]&writeen,
392                                  8'b00000001,bitlinewrite,row8);
393
394           always@(*)
395              casez (wordline)
396                 8'b1???????: bitlinereadz = row1;
397                 8'b?1??????: bitlinereadz = row2;
398                 8'b??1?????: bitlinereadz = row3;
399                 8'b???1????: bitlinereadz = row4;
400                 8'b????1???: bitlinereadz = row5;
401                 8'b?????1??: bitlinereadz = row6;
402                 8'b??????1?: bitlinereadz = row7;
403                 8'b???????1: bitlinereadz = row8;
404                 default: bitlinereadz = 8'bxxxxxxxx;
405              endcase
406
407           assign bitlineread = readen ? bitlinereadz : 8'bzzzzzzzz;
408        endmodule
409
410        // The bitselector module determines which set of three bits should be
411        // pulled from the row that was read from memory. The correct three bits
412        // are the bit currently being looked at, as well as the two to its left
413        // and right. If the current bit is located on the left or right edge, the
414        // first or third bit are set to zero, respectively. The zerorow input
415        // indicates that the row being looked at is outside the indexing of the
416        // matrix. In these case all three bits are set to zero.
417        module bitselector(input [7:0] en,
418                           input zerorow,
419                           input [7:0] d,
420                           output [2:0] q);
421
422           reg [2:0] m;
423
424           always@(*)
425              casez (en)
426                 8'b1???????: m = {1'b0,d[7:6]};
427                 8'b?1??????: m = d[7:5];
```

```
428                8'b??1?????: m = d[6:4];
429                8'b???1????: m = d[5:3];
430                8'b????1???: m = d[4:2];
431                8'b?????1??: m = d[3:1];
432                8'b??????1?: m = d[2:0];
433                8'b???????1: m = {d[1:0],1'b0};
434                default: m = 3'bxxx;
435             endcase
436
437          assign q = m&{~zerorow,~zerorow,~zerorow};
438       endmodule
439
440       // The holdmem module holds the sets of three bits outputted by the
441       // bitselector module until all nine bits are available, so that the
442       // correct input is provided to the output logic module.
443       module holdmem(input ph1,ph2,
444                      input reset,
445                      input en,
446                      input [2:0] row,
447                      output cellbit,
448                      output [7:0] eightbits);
449          wire [2:0] row1, row2, row3;
450
451          flopenr #3 reg1(ph1,ph2,reset,en,row,row1);
452          flopenr #3 reg2(ph1,ph2,reset,en,row1,row2);
453          flopenr #3 reg3(ph1,ph2,reset,en,row2,row3);
454
455          assign cellbit = row2[1];
456          assign eightbits = {row1,row2[2],row2[0],row3};
457       endmodule
458
459       // The outputlogic module takes in the bit whose new value is being
460       // calculated, as well as the surrounding eight bits, and calculates
461       // the new cell value.
462       module outputlogic(input cellbit,
463                          input [7:0] eightbits,
464                          output newcellvalue);
465
466          wire [3:0] sum;
467
468          assign sum = eightbits[7]+eightbits[6]+eightbits[5]+eightbits[4]+
469                       eightbits[3]+eightbits[2]+eightbits[1]+eightbits[0];
470          assign newcellvalue = ~(sum[3]|sum[2])&sum[1]&(~sum[0]&cellbit|sum[0]);
471       endmodule
472
473       // The tempmem module holds the new cell values outputted by outputlogic
474       // until the old values are no longer needed.
475       module tempmem(input ph1,ph2,
476                      input reset,
477                      input newbiten,
478                      input finalrow,
479                      input q,
480                      output [7:0] row2);
481
482          wire [15:0] tworows;
483
484          flopenr #16 reg2(ph1,ph2,reset,newbiten,{tworows[14:0],q},tworows);
485
486          assign row2 = finalrow? tworows[7:0]: tworows[15:8];
487       endmodule
488
```

```
489         // The tristate module is only used in the memory module.
490         module tristate(input en,
491                         input [2:0] d,
492                         output [2:0] q);
493            assign q = en ? d : 3'bzzz;
494         endmodule
495
496         module flop #(parameter WIDTH = 8)
497                      (input ph1, ph2,
498                       input [WIDTH-1:0] d,
499                       output [WIDTH-1:0] q);
500
501           wire [WIDTH-1:0] mid;
502
503           latch #(WIDTH) master(ph2, d, mid);
504           latch #(WIDTH) slave(ph1, mid, q);
505         endmodule
506
507         module flopen #(parameter WIDTH = 8)
508                        (input ph1, ph2, en,
509                         input [WIDTH-1:0] d,
510                         output [WIDTH-1:0] q);
511
512           wire [WIDTH-1:0] d2;
513
514           mux2 #(WIDTH) enmux(q, d, en, d2);
515           flop #(WIDTH) f(ph1, ph2, d2, q);
516         endmodule
517
518         module flopenr #(parameter WIDTH = 8)
519                         (input ph1, ph2, reset, en,
520                          input [WIDTH-1:0] d,
521                          output [WIDTH-1:0] q);
522
523           wire [WIDTH-1:0] d2, resetval;
524
525           assign resetval = 0;
526
527           mux3 #(WIDTH) enrmux(q, d, resetval, {reset, en}, d2);
528           flop #(WIDTH) f(ph1, ph2, d2, q);
529         endmodule
530
531         module flopenrval #(parameter WIDTH = 8)
532                         (input ph1, ph2, reset, en,
533                          input [WIDTH-1:0] resetval,
534                          input [WIDTH-1:0] d,
535                          output [WIDTH-1:0] q);
536
537           wire [WIDTH-1:0] d2;
538
539           mux3 #(WIDTH) enrmux(q, d, resetval, {reset, en}, d2);
540           flop #(WIDTH) f(ph1, ph2, d2, q);
541         endmodule
542
543         module latch #(parameter WIDTH = 8)
544                       (input ph,
545                        input [WIDTH-1:0] d,
546                        output reg [WIDTH-1:0] q);
547
548           always@(*)
549             if (ph) q <= d;
```

```
550         endmodule
551
552         module mux2 #(parameter WIDTH = 8)
553                      (input [WIDTH-1:0] d0, d1,
554                       input s,
555                       output [WIDTH-1:0] y);
556
557           assign y = s ? d1 : d0;
558         endmodule
559
560         module mux3 #(parameter WIDTH = 8)
561                      (input [WIDTH-1:0] d0, d1, d2,
562                       input [1:0] s,
563                       output reg [WIDTH-1:0] y);
564
565           always@(*)
566             casez (s)
567               2'b00: y = d0;
568               2'b01: y = d1;
569               2'b1?: y = d2;
570             endcase
571         endmodule
572
```

A3.2: Fixed Netlist for Latch

```
// Library - muddlib10, Cell - latch_c_1x, View - cmos_sch
// LAST TIME SAVED: Feb  4 13:25:38 2010
// NETLIST TIME: Apr  3 19:32:53 2010
`timescale 1ns / 100ps

module latch_c_1x ( q, d, ph );
output  q;

input  d, ph;
trireg    masterb;

//ADDED VARIABLE
reg rst;


specify
   specparam CDS_LIBNAME  = "muddlib10";
   specparam CDS_CELLNAME = "latch_c_1x";
   specparam CDS_VIEWNAME = "cmos_sch";
endspecify


//THESE LINES ARE CHANGED FROM ORIGINAL LATCH
initial begin
 rst = 1; #5; rst = 0;
end

nmos jamb(masterb, cds_globals.gnd_, rst);
//END CHANGED SECTION

rnmos #(0.1)  M18( net030, masterb, phb);
rpmos #(0.1)  M15( net054, masterb, phbuf);
`switch default
nmos #(0.1)   (* const real width = 2.100, length = 0.600; *) M34( phb,
    cds_globals.gnd_, ph);
`switch default
nmos #(0.1)   (* const real width = 2.100, length = 0.600; *) M26( q,
    cds_globals.gnd_, masterb);
`switch default
nmos #(0.1)   (* const real width = 2.100, length = 0.600; *) M35(
    phbuf, cds_globals.gnd_, phb);
`switch default
nmos #(0.1)   (* const real width = 1.200, length = 0.600; *) M14(
    cds_globals.gnd_, net030, master);
`switch default
nmos #(0.1)   (* const real width = 1.800, length = 0.600; *) M32(
    masterinb, cds_globals.gnd_, d);
`switch default
nmos #(0.1)   (* const real width = 1.800, length = 0.600; *) M2(
    masterb, masterinb, phbuf);
`switch default
nmos #(0.1)   (* const real width = 1.800, length = 0.600; *) M1(
    master, cds_globals.gnd_, masterb);
`switch default
```

```
pmos #(0.1)   (* const real width = 3.000, length = 0.600; *) M30(
    masterinb, cds_globals.vdd_, d);
`switch default
pmos #(0.1)   (* const real width = 3.000, length = 0.600; *) M27( q,
    cds_globals.vdd_, masterb);
`switch default
pmos #(0.1)   (* const real width = 3.000, length = 0.600; *) M33( phb,
    cds_globals.vdd_, ph);
`switch default
pmos #(0.1)   (* const real width = 3.000, length = 0.600; *) M36(
    phbuf, cds_globals.vdd_, phb);
`switch default
pmos #(0.1)   (* const real width = 1.200, length = 0.600; *) M16(
    cds_globals.vdd_, net054, master);
`switch default
pmos #(0.1)   (* const real width = 1.800, length = 0.600; *) M3(
    masterb, masterinb, phb);
`switch default
pmos #(0.1)   (* const real width = 2.700, length = 0.600; *) M0(
    master, cds_globals.vdd_, masterb);

endmodule
```

Appendix 4:

Testing Materials

```
1   `timescale 1ns / 1ps
2
3   //////////////////////////////////////////////////////////////////////////////////
4   // Company: Harvey Mudd College
5   // Engineer: Narayan Propato
6   // Create Date:   19:43:51 03/22/2010
7   // Design Name:   gameoflife
8   // Module Name:   C:/NP/Project 2/gameoflife/verilogtestfixture.v
9   // Project Name:  gameoflife
10  // Description: This test fixture simulates user input comprised of setting the
11  // switch values to certain numbers and then pressing enter to input them. These
12  // values are obtained from the testmatrix.txt file. After three iterations, The
13  // values outputted by the chip to the LED matrix are then checked against the
14  // expected ones, which are also located in the testmatrix.txt file.
15  //////////////////////////////////////////////////////////////////////////////////
16
17  module verilogtestfixture;
18
19     // Inputs
20     reg ph1,ph2;
21     reg reset;
22     reg enter;
23     reg [7:0] switch;
24
25     // Outputs
26     wire [7:0] ledpower;
27     wire [7:0] ledcolumn;
28
29     reg [7:0] testmatrix [16:0], currentrow;
30     reg [5:0] vectornum, errors;
31     reg [11:0] cyclea, cycleb;
32     reg ledoutputa,ledoutputb;
33
34     // Instantiate the Unit Under Test (UUT)
35     core dut (
36        .ph1(ph1),
37        .ph2(ph2),
38        .reset(reset),
39        .enter(enter),
40        .switch(switch),
41        .ledpower(ledpower),
42        .ledcolumn(ledcolumn));
43
44     // Initialize test
45       initial
46         begin
47           $readmemb("testmatrix.txt", testmatrix);
48           vectornum = 0; errors = 0;
49           ph1 = 0;
50           ph2 = 0;
51           enter = 0;
52           switch = 0;
53           cyclea = 0;
54           cycleb = 0;
55           reset <= 1; # 11; reset <= 0;
56
57           #36;   enter <= 1; switch <= testmatrix[vectornum];
58           vectornum = vectornum + 1; #12; enter <= 0;
59           #1500; enter <= 1; switch <= testmatrix[vectornum];
60           vectornum = vectornum + 1; #12; enter <= 0;
61           #1500; enter <= 1; switch <= testmatrix[vectornum];
```

```
62                vectornum = vectornum + 1; #12; enter <= 0;
63                #1500; enter <= 1; switch <= testmatrix[vectornum];
64                vectornum = vectornum + 1; #12; enter <= 0;
65                #1500; enter <= 1; switch <= testmatrix[vectornum];
66                vectornum = vectornum + 1; #12; enter <= 0;
67                #1500; enter <= 1; switch <= testmatrix[vectornum];
68                vectornum = vectornum + 1; #12; enter <= 0;
69                #1500; enter <= 1; switch <= testmatrix[vectornum];
70                vectornum = vectornum + 1; #12; enter <= 0;
71                #1500; enter <= 1; switch <= testmatrix[vectornum];
72                vectornum = vectornum + 1; #12; enter <= 0;
73
74                #1500; enter <=1; switch <= 8'h00; #12; enter <=0;
75                #3000; enter <=1; switch <= 8'h00; #12; enter <=0;
76                #3000; enter <=1; switch <= 8'h00; #12; enter <=0;
77              end
78
79
80             // Generate clock to sequence tests
81             always
82               begin
83                ph1 <= 0; ph2 <= 0; #1;
84                ph1 <= 1; cyclea <= cyclea + 1; # 4;
85                ph1 <= 0; #1;
86                ph2 <= 1; cycleb <= cycleb + 1; # 4;
87               end
88
89             // ledoputa and ledoutputb are used to identify when a
90             // new row should be checked
91             always @(posedge ph1)
92                 case (cyclea)
93                    12'h886: ledoutputa = 1;
94                    12'h896: ledoutputa = 1;
95                    12'h8a6: ledoutputa = 1;
96                    12'h8b6: ledoutputa = 1;
97                    12'h8c6: ledoutputa = 1;
98                    12'h8d6: ledoutputa = 1;
99                    12'h8e6: ledoutputa = 1;
100                   12'h8f6: ledoutputa = 1;
101                   12'h906: ledoutputa = 1;
102                   default: ledoutputa = 0;
103                endcase
104
105            always @(posedge ph2)
106                case (cycleb)
107                   12'h886: ledoutputb = 1;
108                   12'h896: ledoutputb = 1;
109                   12'h8a6: ledoutputb = 1;
110                   12'h8b6: ledoutputb = 1;
111                   12'h8c6: ledoutputb = 1;
112                   12'h8d6: ledoutputb = 1;
113                   12'h8e6: ledoutputb = 1;
114                   12'h8f6: ledoutputb = 1;
115                   12'h906: ledoutputb = 1;
116                   default: ledoutputb = 0;
117                endcase
118
119            always @(posedge ledoutputa) begin
120               currentrow = testmatrix[vectornum];
121            end
122
```

```
123            // check if test was sucessful and apply next one
124            always @(posedge ledoutputb) begin
125                   if (currentrow === 8'bxxxxxxxx) begin
126                 $display("Completed the test with %d errors.",
127                          errors);
128                 $stop;
129               end
130               else if ((ledcolumn !== currentrow)) begin
131                  $display("Error: output mismatches as %h (%h expected)",
132                           ledcolumn, currentrow);
133                  errors = errors + 1;
134               end
135               vectornum = vectornum + 1;
136            end
137         endmodule
138
```

A4.2: Test Vectors

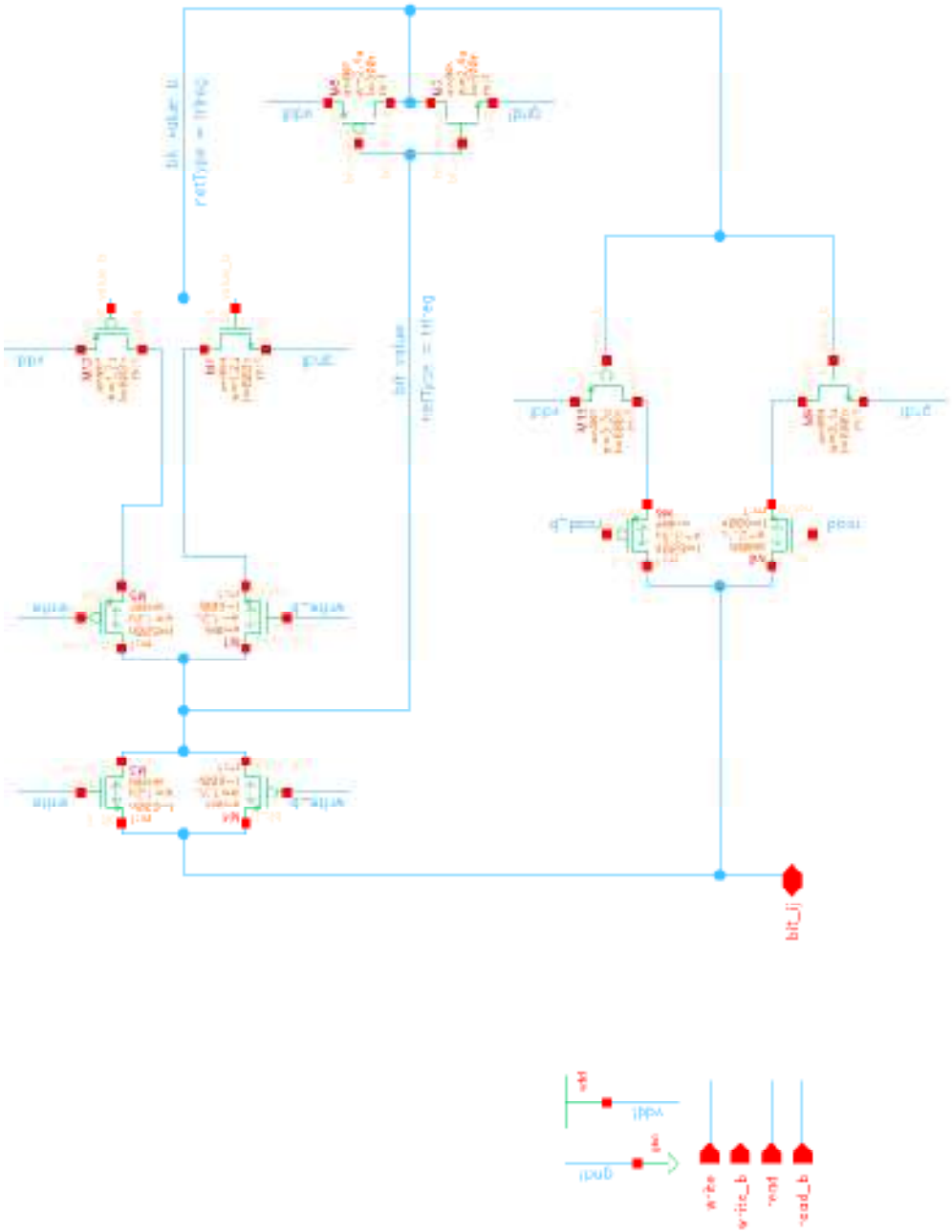
```
00000000        11111111        10011101
00000000        00000000        00110000
00000000        11111111        11011111
00000000        00000000        11011011
00000000        11111111        00111100
00000000        00000000        10110110
00000000        11111111        11111111
00000000        00000000        11001001
00000000        00011000        00100000
00000000        00000000        11000000
00000000        00011000        00000011
00000000        00000000        00100100
00000000        00011000        11000100
00000000        00111100        00000100
00000000        00000000        00000011
00000000        00000000        00000000
xxxxxxxx        xxxxxxxx        xxxxxxxx

11111111        11111001
11111111        10001001
11111111        11011001
11111111        11110000
11111111        01111101
11111111        00000110
11111111        01111110
11111111        10100110
00000000        01011010
00000000        01000001
00000000        00011001
00000000        00001010
00000000        00001000
00000000        01010111
00000000        10000011
00000000        01010000
xxxxxxxx        xxxxxxxx
```

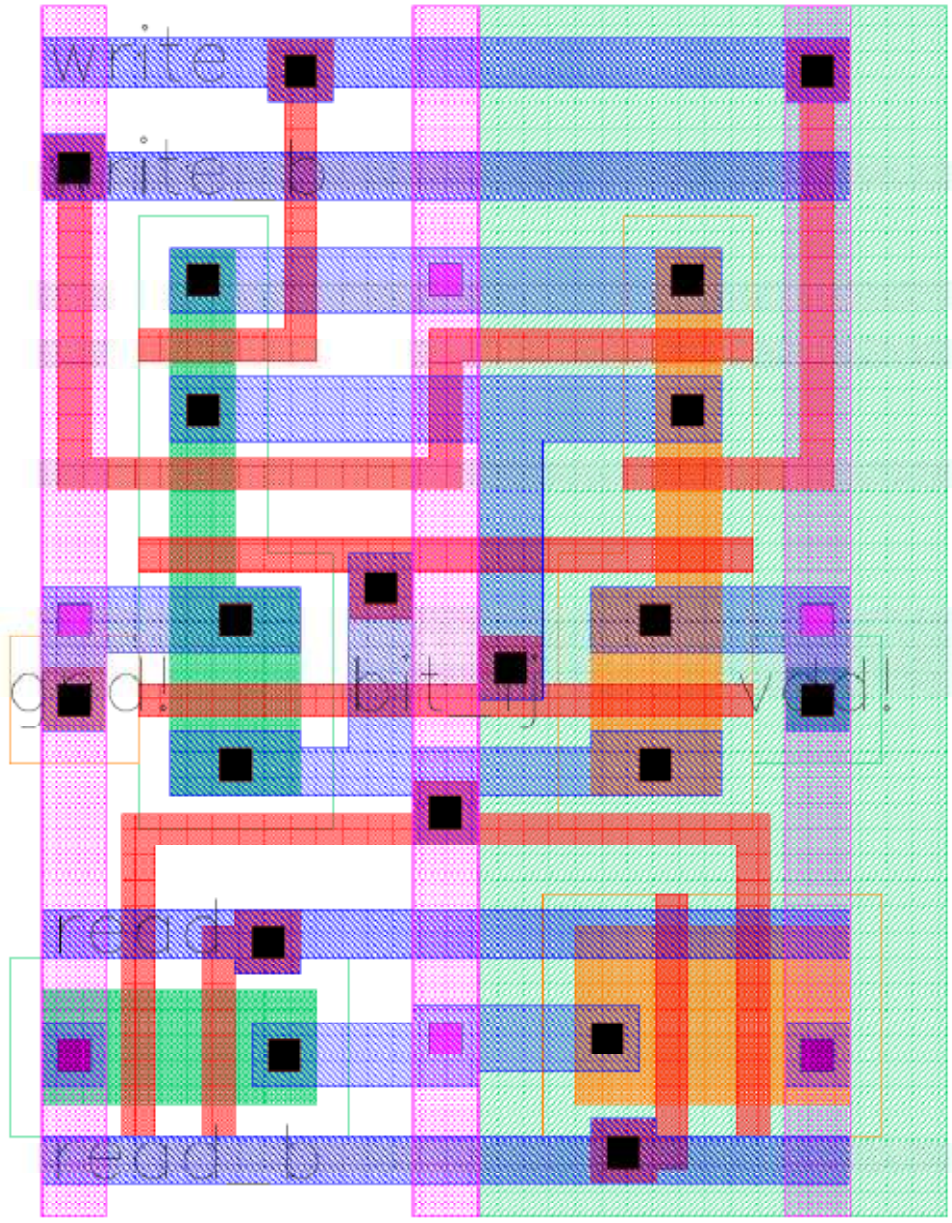
Appendix 5:

Schematics and Layout

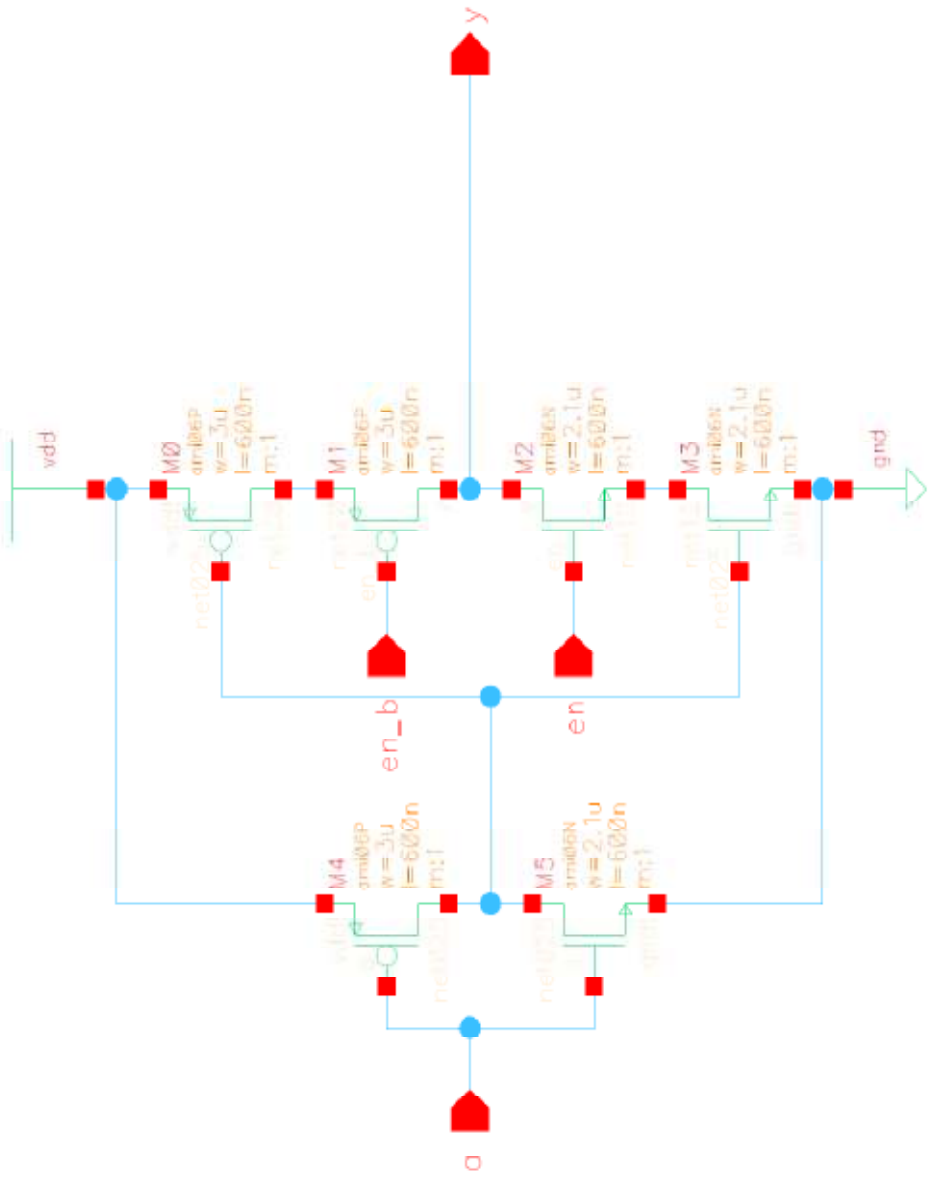
A5.1.1: 12t_cell Schematic



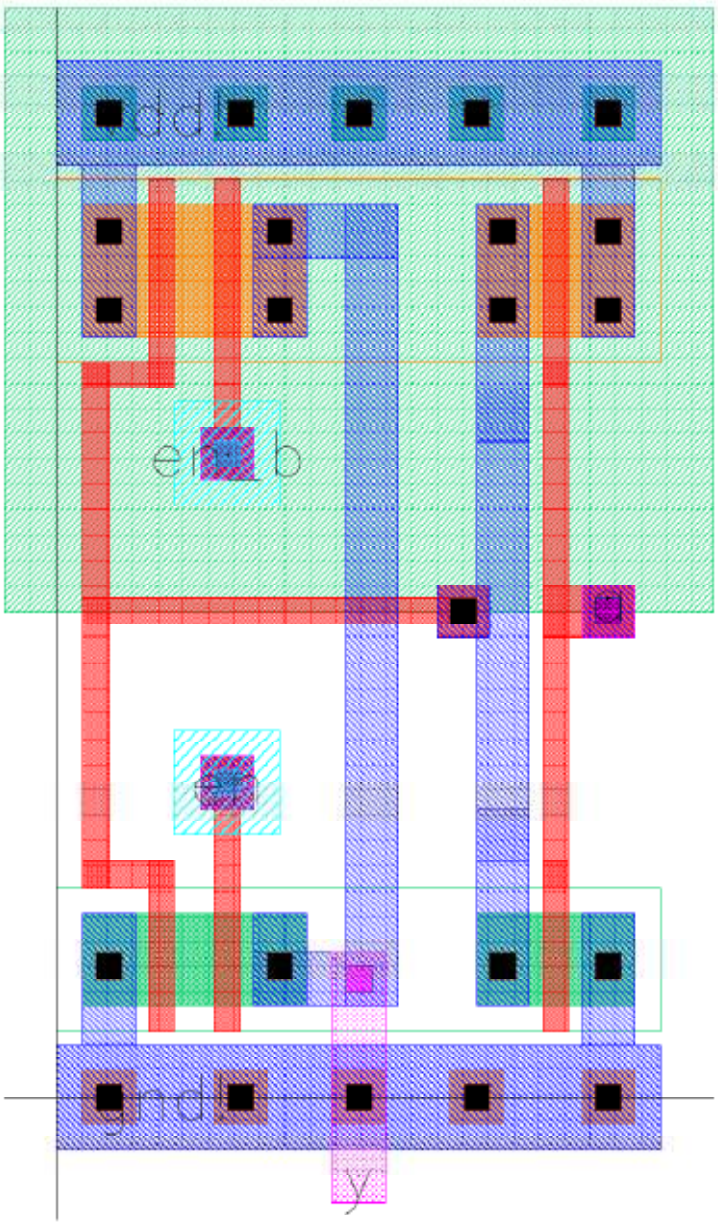
A5.1.1: 12t_cell Layout



A5.2.1: tri_buf Schematic

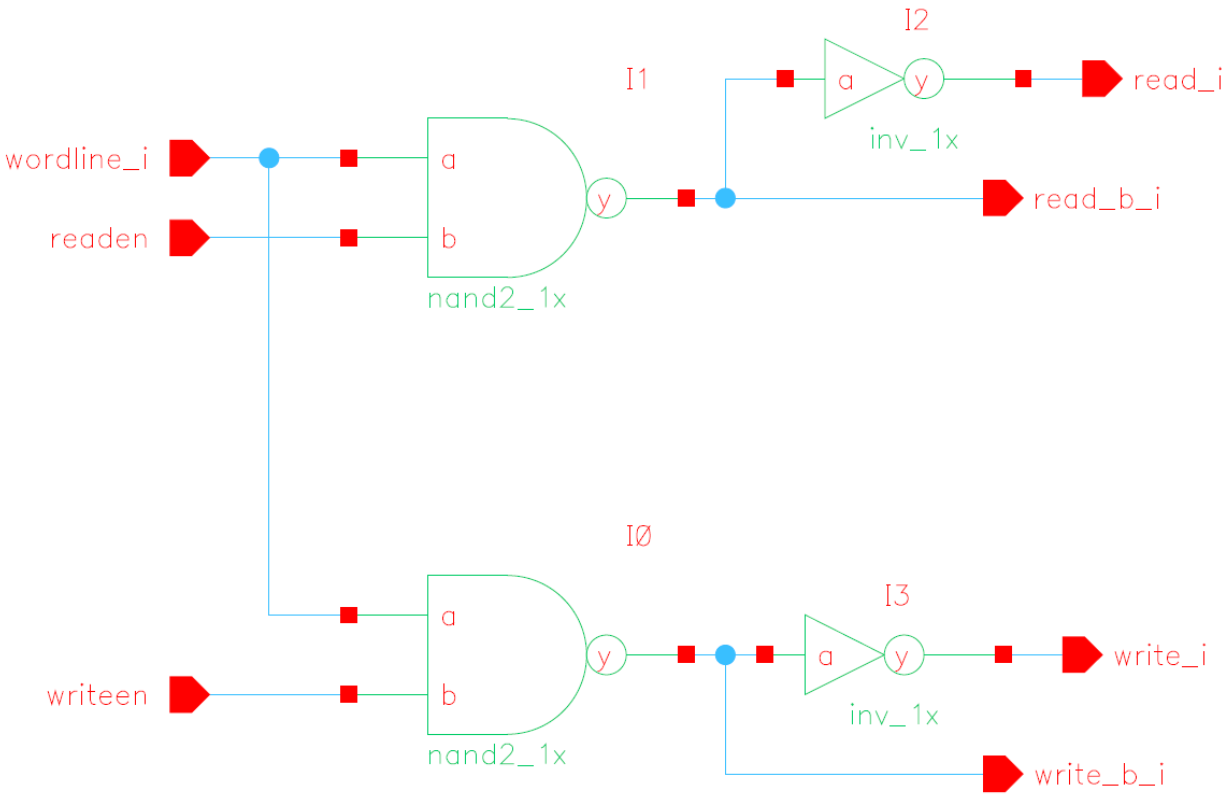


A5.2.2: tri_buf Layout

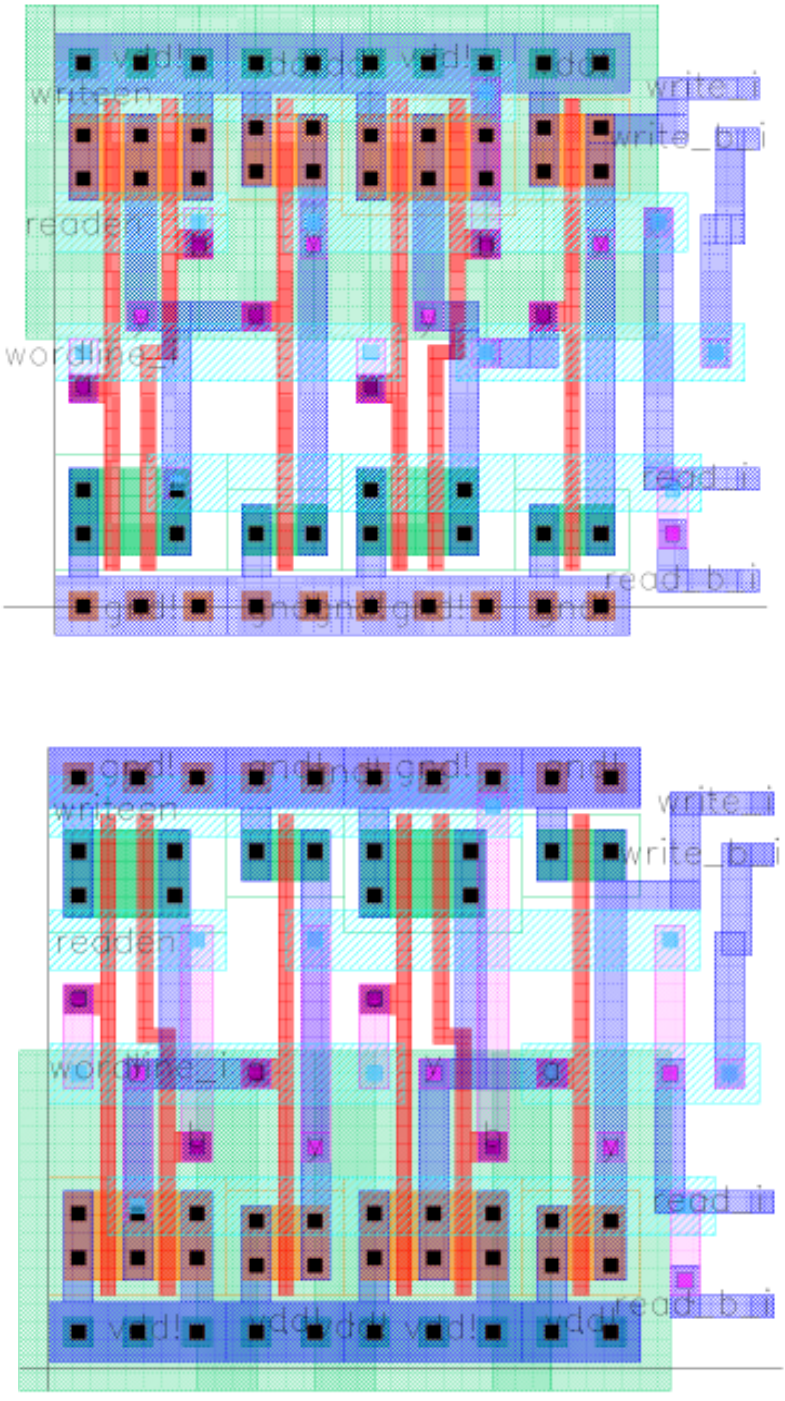


A5.3.1: wordline_bitcell and wordline_bitcell_odd Schematics

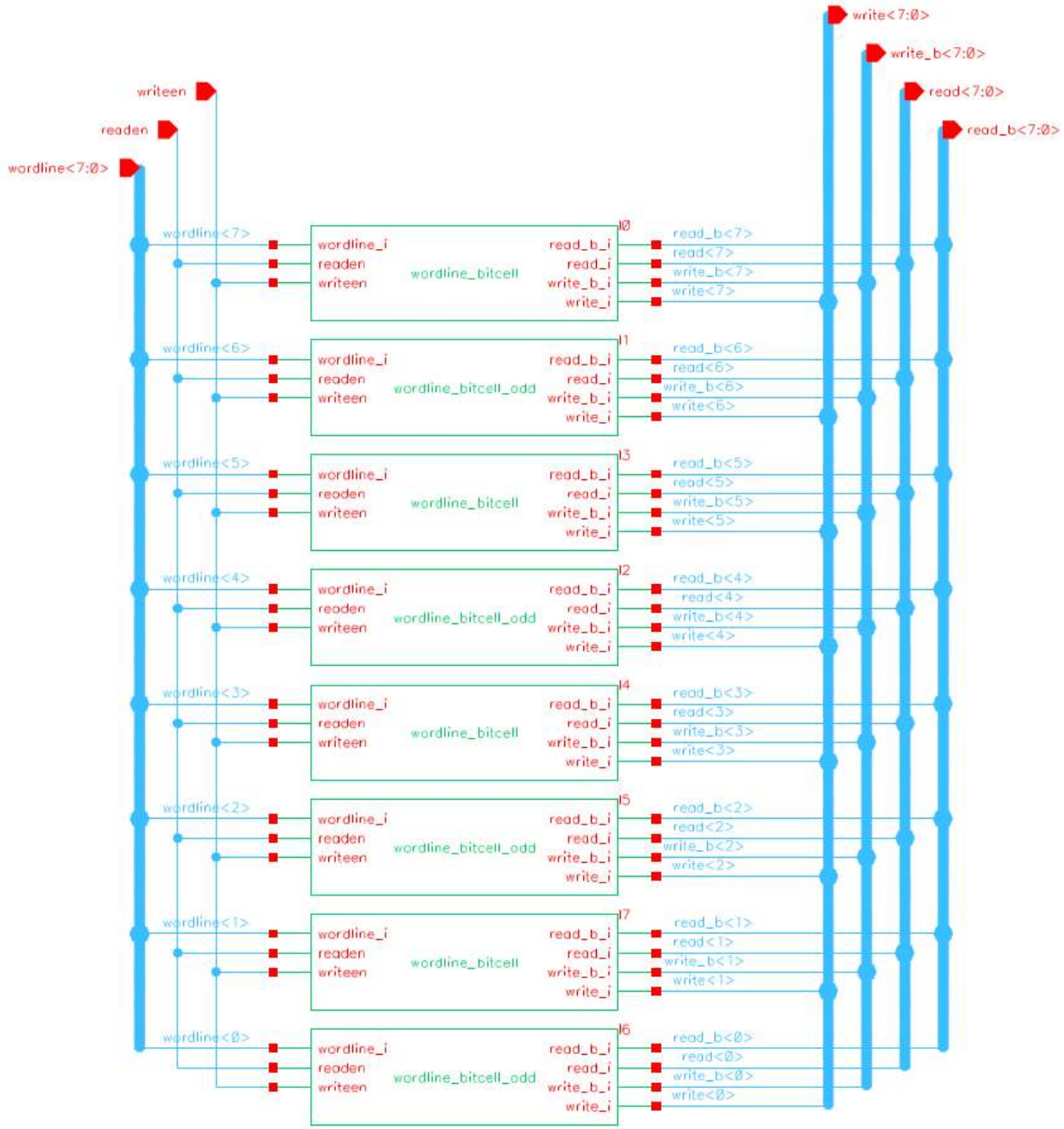
Both cells share the same schematic



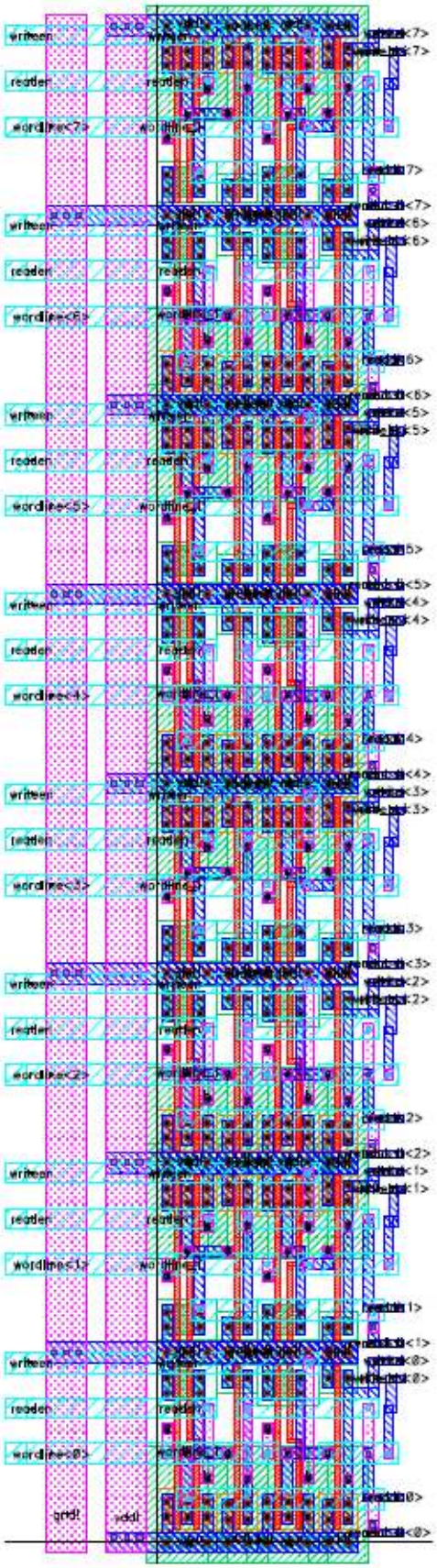
A5.3.2: wordline_bitcell (top) and wordline_bitcell_odd (bottom) Layouts



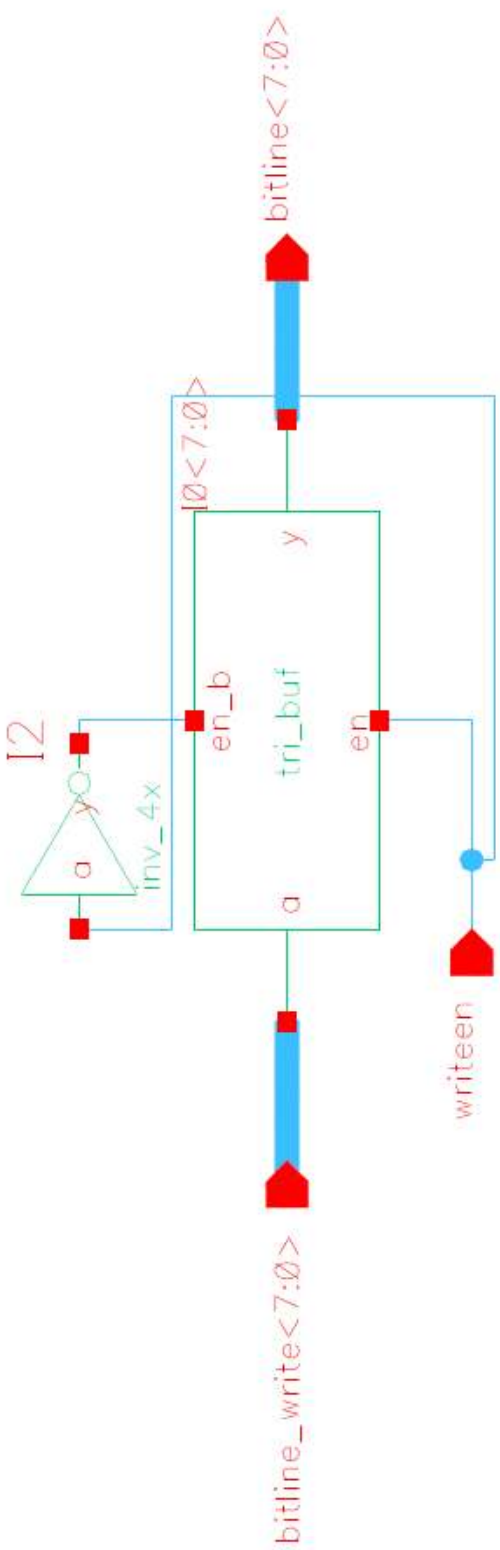
A5.4.1: wordline Schematic

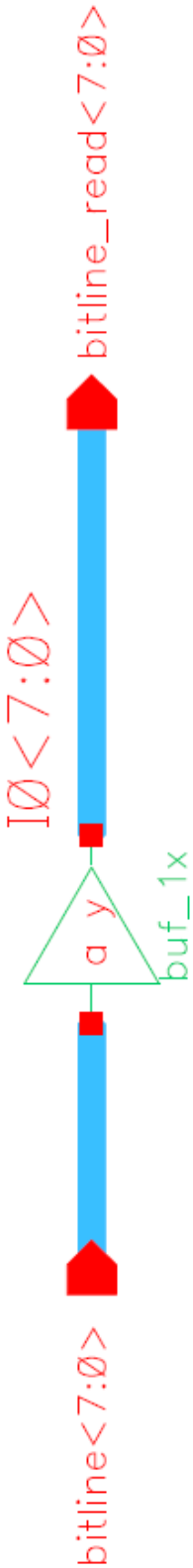


A5.4.2: wordline Layout

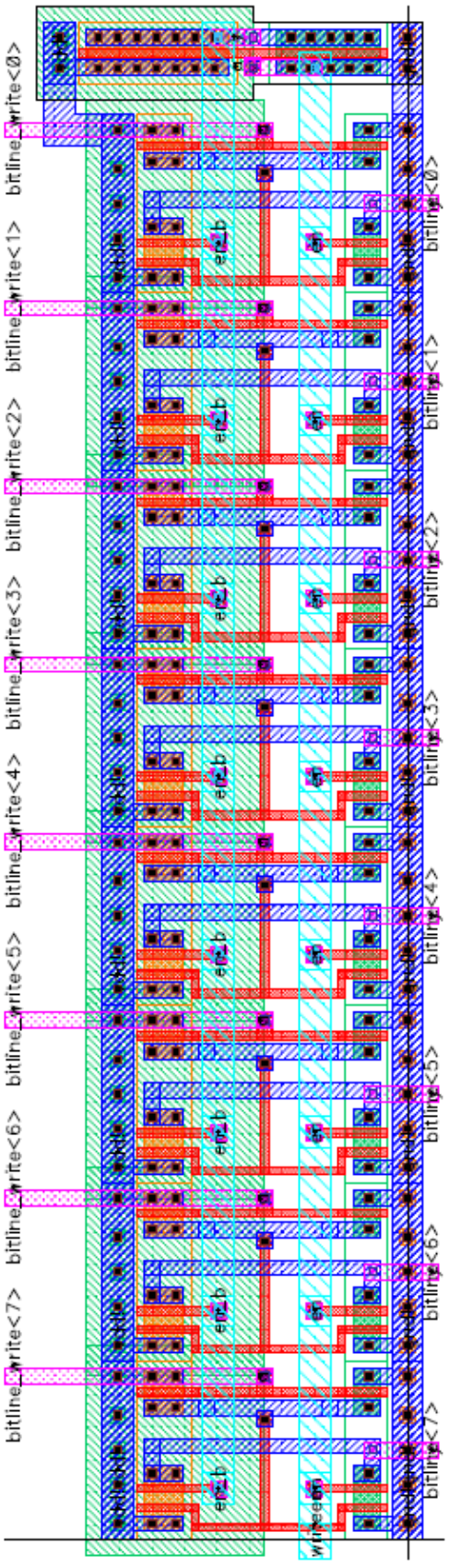


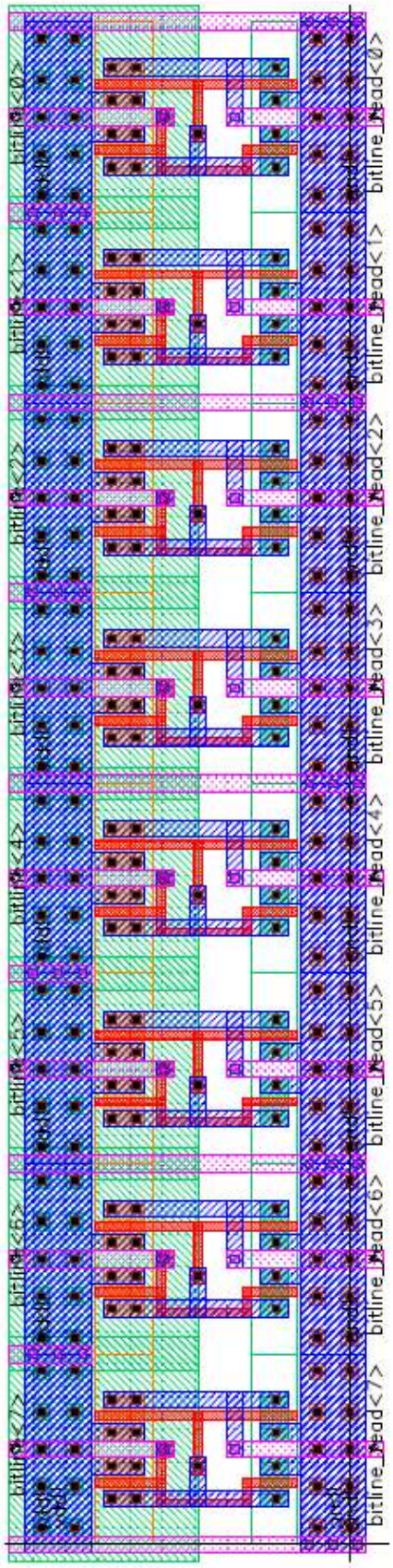
A5.5.1: bitline_write (left) and bitline_read (right) Schematics



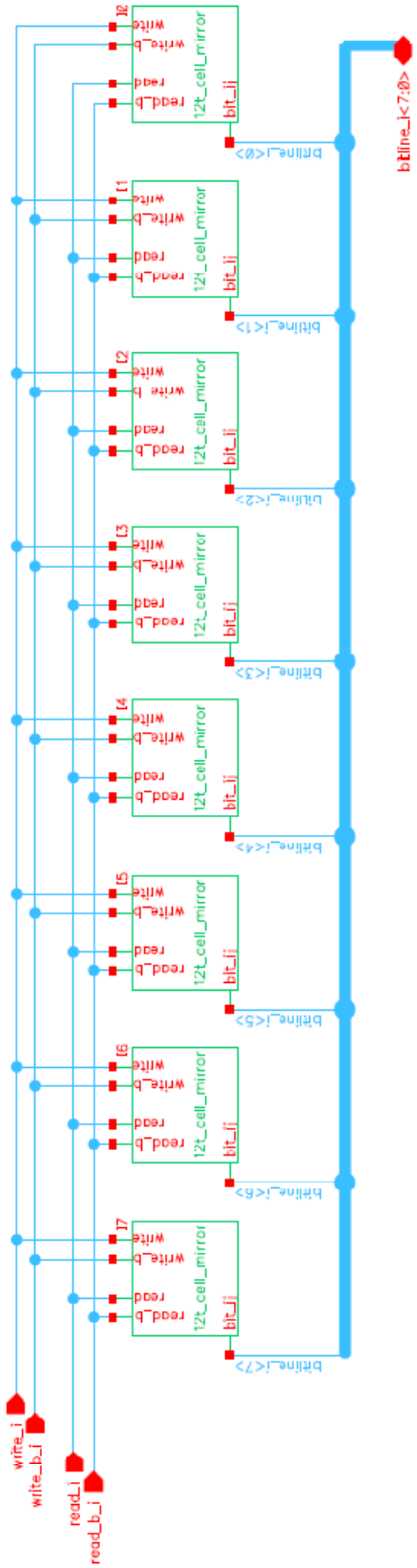


A5.5.2: bitline_write (left) and bitline_read (right) Layouts

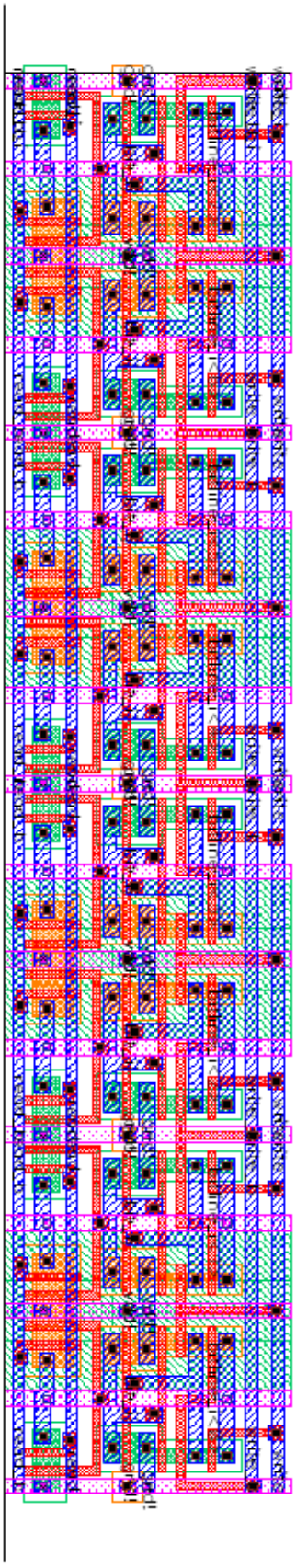




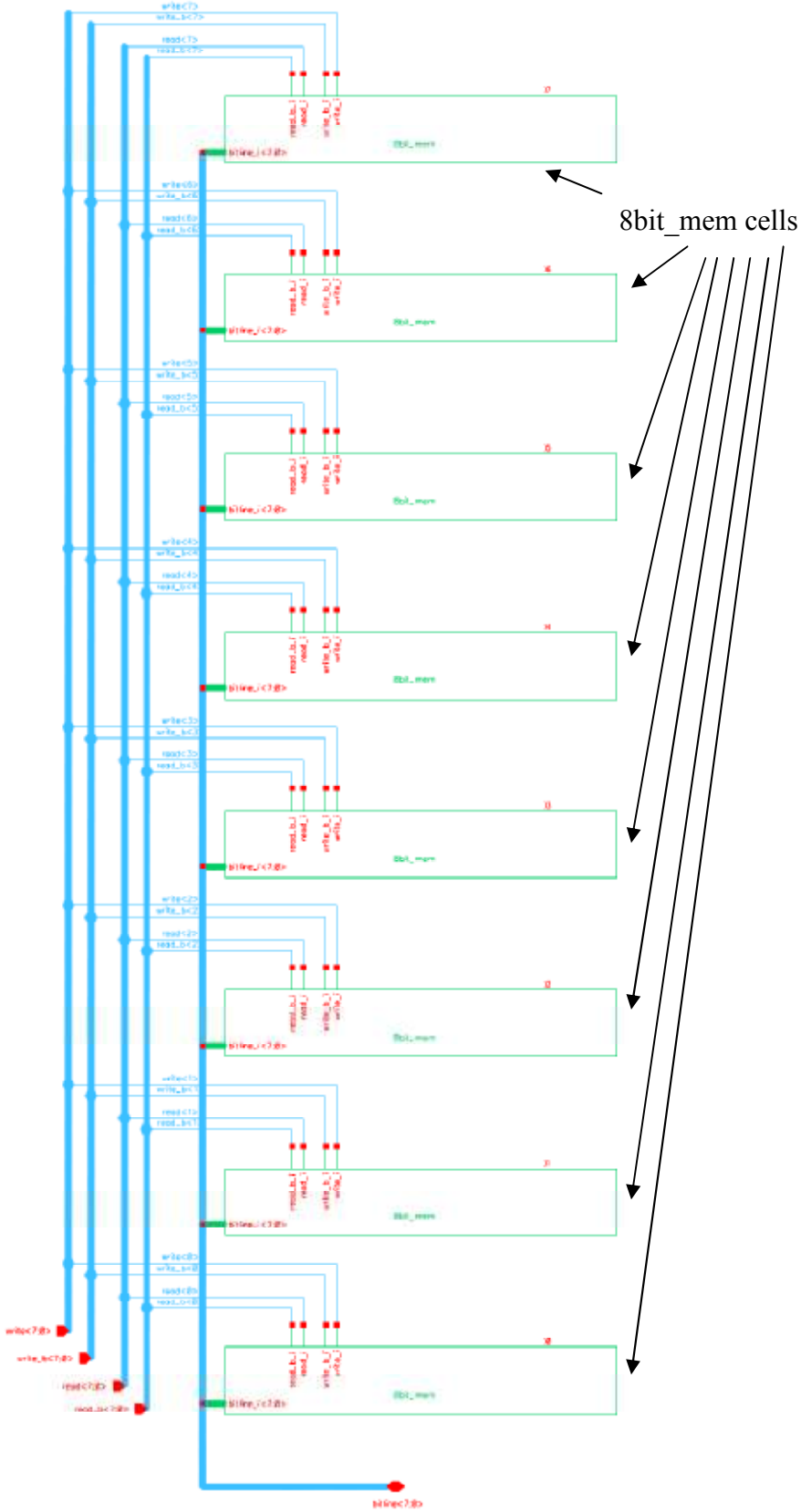
A5.6.1: 8bit_mem Schematic



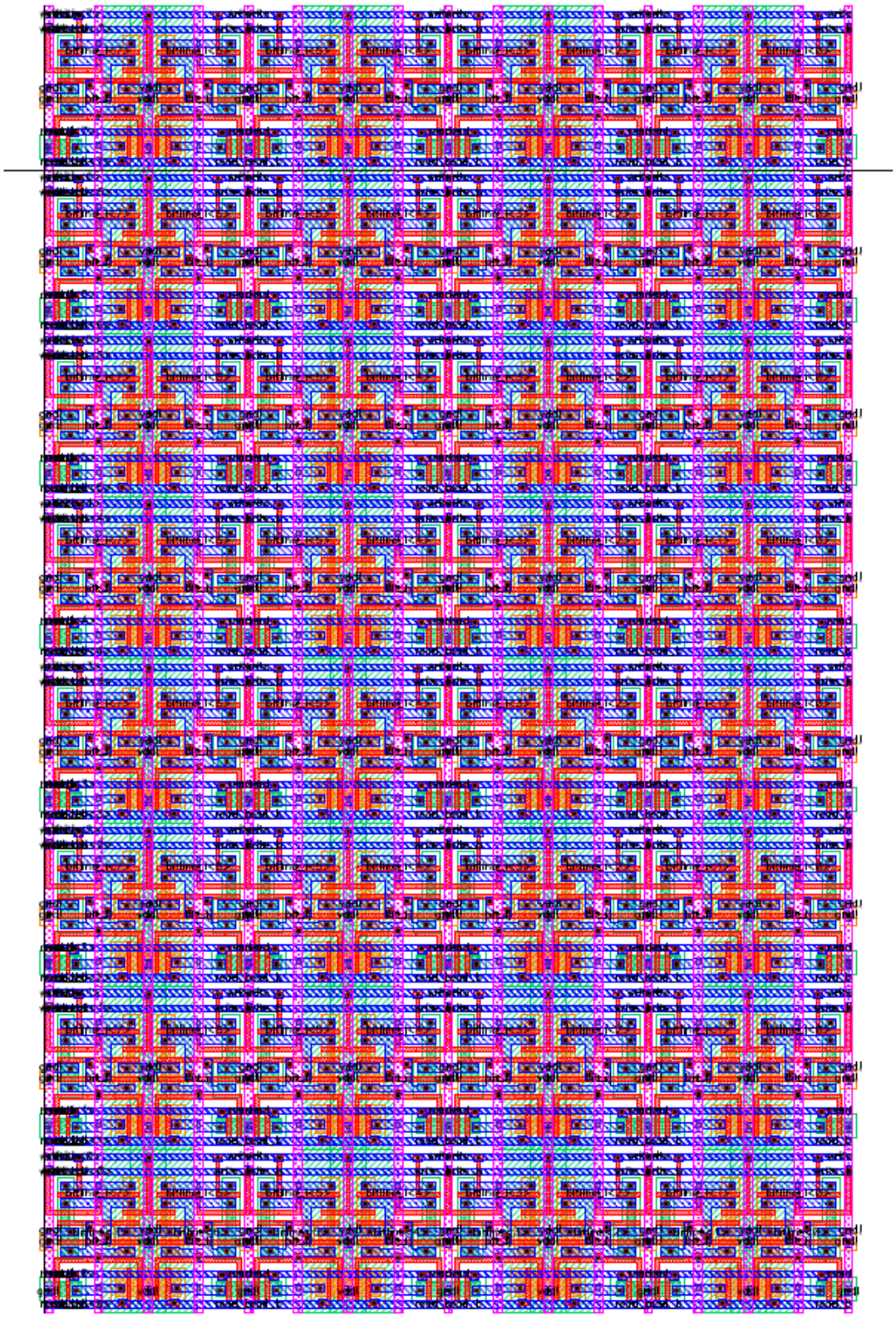
A5.6.2: 8bit_mem Layout



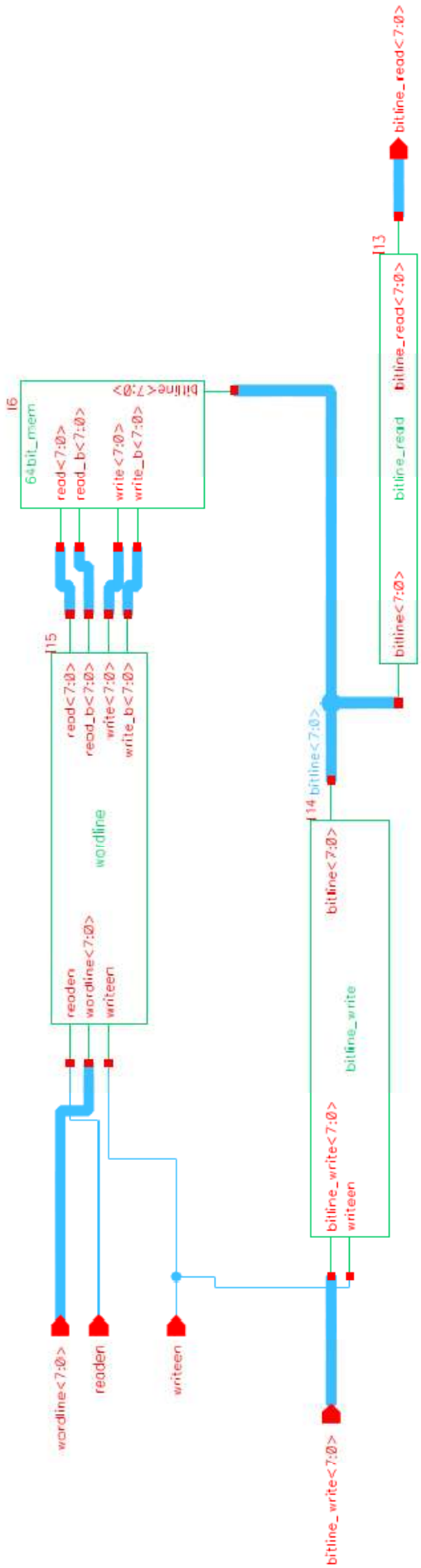
A5.7.1: 64bit_mem Schematic



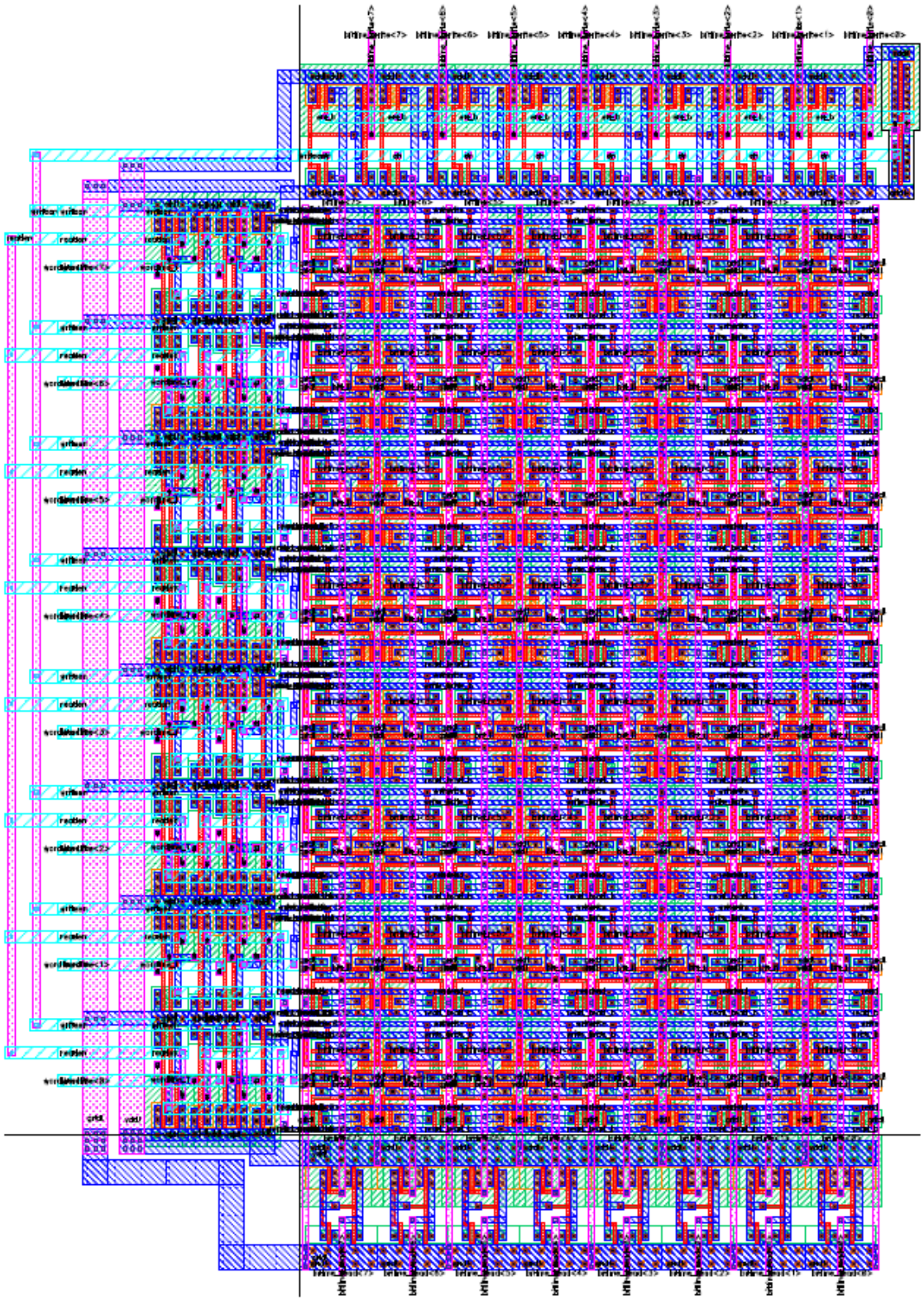
A5.7.2: 64bit_mem Layout



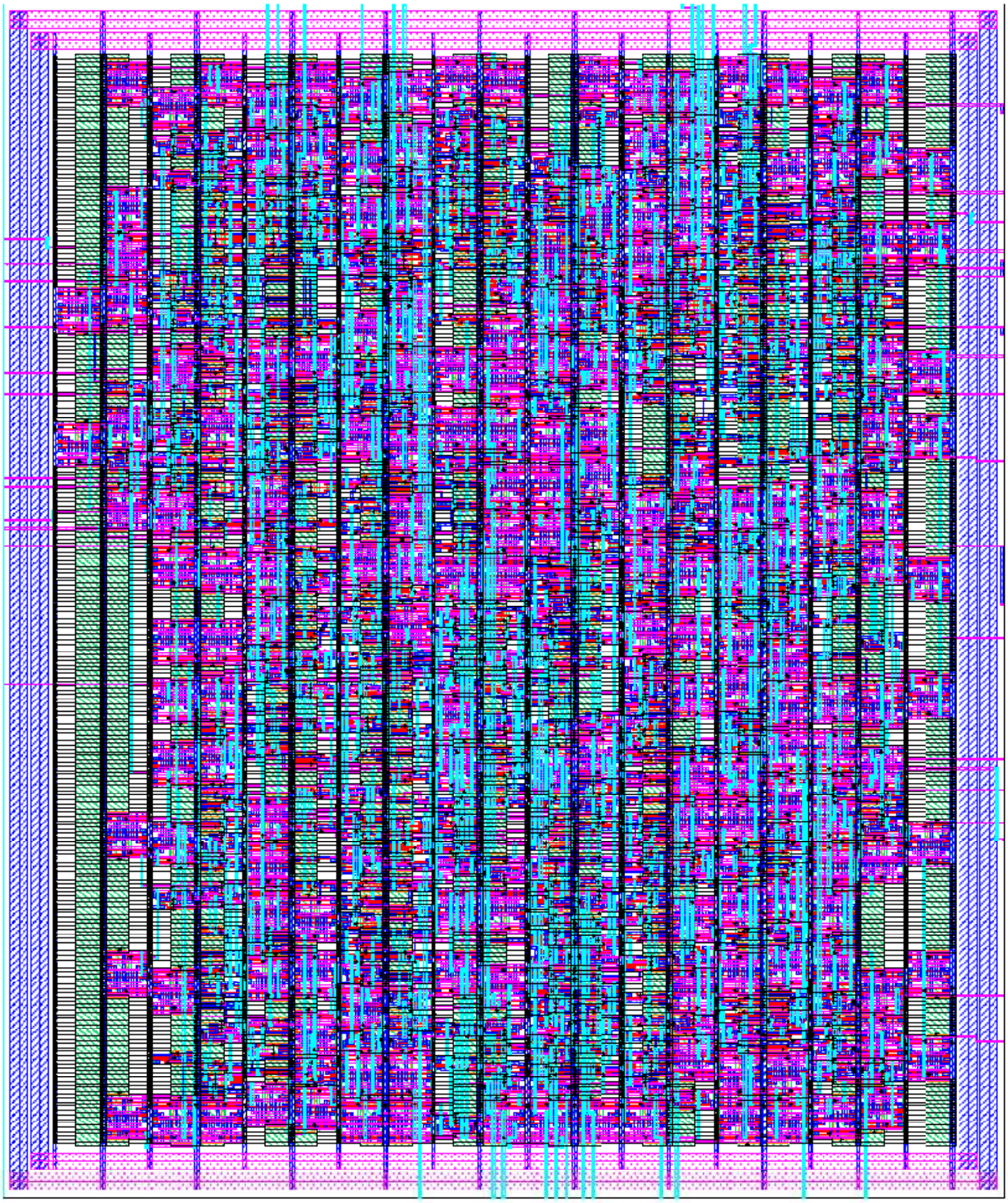
A5.8.1: register Schematic



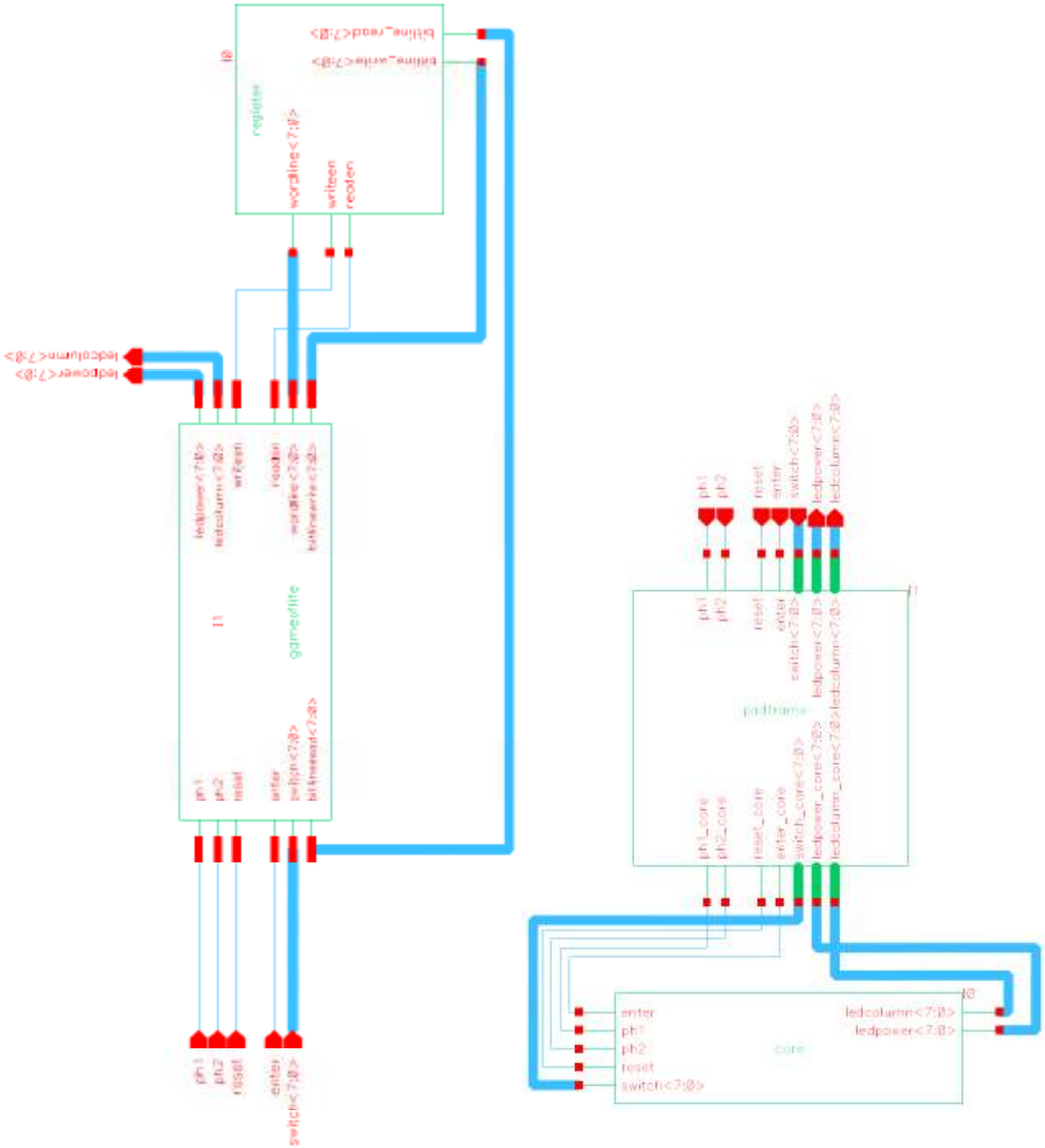
A5.8.2: register Layout



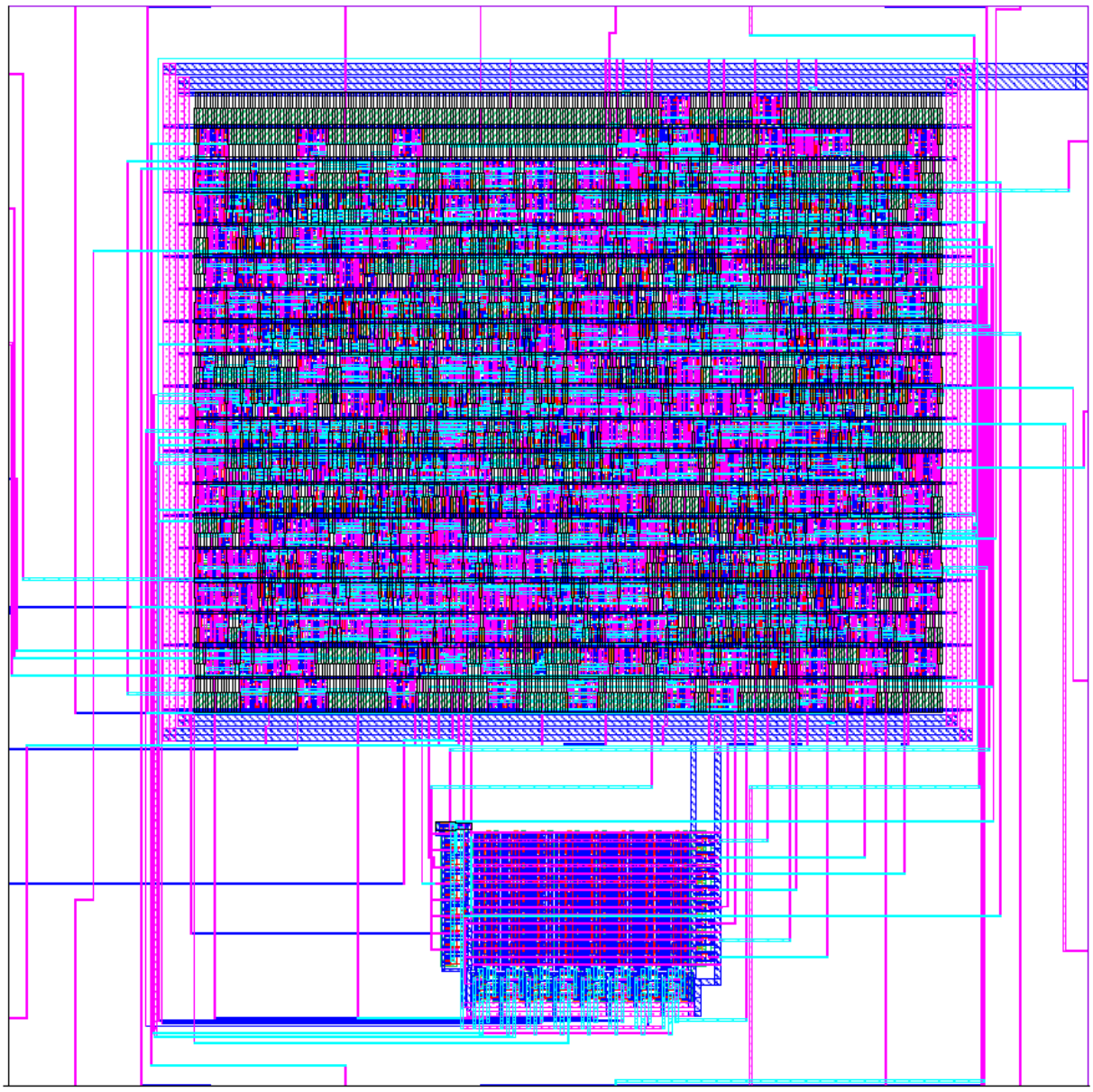
A5.9: gameoflife Layout



A5.10.1: core (left) and chip (right) Schematics



A5.10.2: core Layout



A5.10.3: chip Layout